St. Cloud State University

The Repository at St. Cloud State

Culminating Projects in Computer Science and Information Technology

Department of Computer Science and Information Technology

12-2023

Developing Blind-Bidding Auctions to Explore Fully Homomorphic Encryption

Adeline Moll

Follow this and additional works at: https://repository.stcloudstate.edu/csit_etds

Part of the Computer Sciences Commons

Recommended Citation

Moll, Adeline, "Developing Blind-Bidding Auctions to Explore Fully Homomorphic Encryption" (2023). *Culminating Projects in Computer Science and Information Technology*. 47.
https://repository.stcloudstate.edu/csit_etds/47

This Thesis is brought to you for free and open access by the Department of Computer Science and Information Technology at The Repository at St. Cloud State. It has been accepted for inclusion in Culminating Projects in Computer Science and Information Technology by an authorized administrator of The Repository at St. Cloud State. For more information, please contact tdsteman@stcloudstate.edu.

Developing Blind-Bidding Auctions to Explore Fully Homomorphic Encryption

by

Adeline Moll

A Thesis

Submitted to the Graduate Faculty of

St. Cloud State University

in Partial Fulfillment of the Requirements

for the Degree

Master of Science in

Computer Science

December 2023

Thesis Paper Committee:
Akalanka Mailewa, Chairperson
Dale Buske
Mark Schmidt
Aleksandar Tomovic

Abstract

In blind-bidding auctions, determining the winner requires processing bids without revealing non-winning bid information. This study explores the use of Fully Homomorphic Encryption (FHE) as a solution for this challenge. FHE permits operations like addition and multiplication on encrypted data, enabling potentially arbitrary computations without exposing underlying values. Hower, leveraging FHE for blind-bidding auctions is not straightforward. Classic sorting algorithms are not directly applicable due to FHE's constraints. This thesis presents our approach to designing a blind-bidding auction using FHE, focusing specifically on determining the maximum bid within the given computational boundaries.

Acknowledgements

I would like to thank Dr. Akalanka Mailewa for his time and input as the chair of my thesis committee. I would also like to thank the other members of my thesis committee, Dr. Dale Buske, Dr. Mark Schmidt, and Dr. Alek Tomovic. I would like to thank St. Cloud State University, the Department of Computer Science, and all the professors in said department.

Table of Contents

Page

List of Tables.....6

List of Figures.....7

Chapter

Chapter 1: Introduction.....8

1.1 Explanation of Blind Bidding.....8

1.2 Fully Homomorphic Encryption.....8

1.3 Objectives of the Study.....9

1.4 Research Questions.....10

Chapter 2: Literature Review: Background of Homomorphic Encryption.....12

2.1 Theoretical Introduction to Privacy Homomorphisms.....12

2.1.1 Mathematical Representation of a Homomorphism.....13

2.1.2. Example of a Theoretical Privacy Homomorphism.....14

2.2 Learning with Errors.....14

2.2.1 Mathematical Basis of LWE.....15

2.2.2 Example of a Simple LWE System of Linear Equations.....16

2.3 Somewhat Homomorphic Encryption.....17

2.4 Gentry's Fully Homomorphic Encryption Scheme.....18

2.4.1. Bootstrapping.....19

2.5 Reductions to the Bootstrapping Cost.....19

Chapter Page

2.5.1 Reducing Bootstrapping Costs with Modulus Switching.. 20

2.5.2 Bootstrapping in Quasilinear Time... 21

2.5.4 TFHE: Fast Fully Homomorphic Encryption over the Torus.................................. 22

Chapter 3: Blind-Bidding Auction..25

3.1. System Requirements..25

3.2. Methods..25

3.2.1 Justification for Tool Selection... 26

3.3 Implementation and Challenges...26

3.3.1 Data Input and Pre-Processing...27

3.3.2 Bidding Logic Implementation... 27

Chapter 4: Results.. 38

4.1 Computation Times..38

4.1.1 Program Initialization for Different Parameters... 38

4.1.2 Influence of Parameters on compute_max_min()...40

4.1.3 Average vs. Theoretical Computation Time of identify_max_bids()...................... 41

4.2 Accuracy.. 44

4.2.1 Fluctuation in Accuracy with Changes in n and i..44

4.2.2 Fluctuation in Accuracy with Changes in Initial Configuration of Bids................. 46

Chapter 5: Discussion.. 49

5.2 Parameter Selection & Accuracy.. 49

Chapter Page

5.2.1 Accuracy Based on Starting Configuration........ 49

5.2.2 Speed........ 54

5.3 Practicality for Use........ 55

Chapter 6: Conclusion........ 56

References........ 57

Appendices........ 61

Appendix A: System Settings, Build, and Compilation........ 61

Appendix B: Cargo.toml........ 61

Appendix C: mair.rs........ 61

Appendix D: create_bid.rs........ 63

Appendix E: libr.s........ 64

List of Tables

Table Page

1. Computation time required to build the KSK and BSK for different sets of parameters....39
2. Computation time required to perform the compute_max_min() function for different sets of parameters.... 41
3. Expected vs. Actual Computation Times for Given n and i Values....43
4. Accuracy of Results of Winning Bids, for Given n and i Values....45
5. Distribution of Comparisons Among Encrypted Values for Different Starting Configurations....47
6. Number of Comparisons for Each Value, with an Ascending Starting Configuration.... 51
7. Number of Comparisons for Each Value, with an Descending Starting Configuration.... 52
8. Initial Configuration of Bids and the Distribution of Comparisons.... 53

List of Figures

Figure Page

1. Number of Comparisons in the identify_max_bids() function based on the total number of bids and the number of tie winning bids for arrays of size 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100 35
2. Number of Comparisons in the identify_max_bids() function based on the total number of bids and the number of tie winning bids for all arrays between 1 and 100 36
3. For n=9, Accuracy of Results for i values between 1 and 9 45
4. Accuracy of Results Based on the Number of Times Compared 48

Chapter 1: Introduction

1.1 Explanation of Blind Bidding

Alice, Bob, and Charlie are taking part in a blind bidding auction. Each one of them aims to make the highest bid. Each of them wants to keep their bid private. Each of them also wants to be confident that the winning bid was selected by the auction owner, without favoritism. The winning bid, and who made that bid, should be made public, but no other information should be determinable. This includes the ordering of the non-winning bids.

1. Everyone should be able to see the value of the winning bid.
2. No information should be revealed about non-winning bids.

With the rise of digital auctions, ensuring privacy and fairness is important. In this study, we use Fully Homomorphic Encryption (FHE) to set up a blind-bidding auction system.

1.2 Fully Homomorphic Encryption

A Homomorphic Encryption scheme is a scheme in which an operation can be performed on encrypted data. Fully Homomorphic Encryption (FHE) schemes are homomorphic encryption schemes that are both multiplicatively and additively homomorphic. Since all circuits can be expressed using only multiplication and addition, a FHE scheme allows for arbitrary computations to be performed on encrypted data.

Homomorphic Encryption was introduced in a theoretical sense by Rivest, Adleman, and Dertouzos in 1978 under the name of Privacy Homomorphisms [1]. They illustrated the types of privacy homomorphisms that might exist, though, as they acknowledged at that time, their examples were weak cryptographically.

The first known implementation of a FHE Scheme was by Gentry [2] in 2008. Gentry's scheme was based on lattices, which support efficient evaluation for circuits of an arbitrary

depth. The scheme was built on the hardness of lattices and learning with errors (LWE) [3, 4]. This first scheme introduced by Gentry involved an expensive bootstrapping step. Since then, many advances have been made to reduce the cost of bootstrapping [5, 6, 7, 8, 9].

Over the years, various techniques and optimizations have been introduced, significantly reducing bootstrapping time, from the original scheme introduced by Gentry in 2008 to the notable reduction to 127 ms in 2017 [2, 10, 11, 12].

In July of 2022 the Zama Team released Concrete by Zama [30]. This open source framework is built on a variant of TFHE [13], and introduces a programmable bootstrapping technique, where a function can be evaluated during the bootstrapping process. The Concrete library is the FHE library used in this project.

As we approach the quantum era, the importance of FHE is becoming more evident. FHE is a candidate for post-quantum security. Quantum computing holds the potential to undermine the security of many current encryption schemes, leaving currency security infrastructure vulnerable. By continuing to advance FHE, we are paving a way for security in the quantum future.

1.3 Objectives of the Study

In this work, we set up a blind-bidding auction system to explore the nuances of FHE. In the auction, bidders are assigned a bidder ID. The bids in the auction encode the bid value as the more significant bits, while the bidder ID is stored in the least significant bits. The auction takes encrypted bids, and outputs the highest bid. In the case of multiple tie bids, all tie bids are output by the system.

The blind-bidding auction system serves as a testbed to explore FHE. The primary objective of this study is to work with FHE data and explore the intricacies of the encryption

scheme and operation upon FHE data. We don't expect that the blind-bidding auction will be practical for actual implementation, but instead expect to develop further insight into the intricacies of FHE.

The reason the blind-bidding auction serves as such an apt testbed for studying FHE is its detailed requirement of needing to reveal a precise amount of information, but nothing more. Additionally, because we don't quite need to sort the full array of bids, but neither can we just compute the maximum, considering the best way to accomplish this provides an interesting problem to tackle in the context of FHE. We need to consider how we can reveal some unknown number of tied maximum bids, without ever decrypting or revealing additional information about non-winning bids.

In this study, we present a sorting system specifically tailored to blind-bidding auctions. Unlike traditional sorting systems, our system is designed to prioritize confidentiality of bids. The primary innovation lies in the ability to identify an undetermined number of highest bids, without revealing any information about the non-winning bids.

The scope of this study is only small bids, ranging between 0-2, and for no more than 100 bids. All computations should be able to be done on an average personal computer. The implementation and testing is specifically on a MacBook Pro with an Apple M1 chip and 8 GB memory, and most tested computation times were under one hour of computation time.

1.4 Research Questions

In implementing a FHE scheme to set up the blind-bidding auction system, the following are studied:

- The impact of initial parameters on the computation time and accuracy.
- How changes to the initial arrangement of a set of bids impacts the accuracy.

- How accurately can we estimate computation times based on the number of bids and potential tied winning values?
- For what sizes of input bids, if any, the blind-bidding auction is practical.

Based on our methodologies, we expect to gain a clearer understanding of how initial parameters impact computation time and accuracy. We also expect to be able to accurately estimate computation time based on the number of comparisons required in the auction.

We aim to not only expand technical knowledge surrounding encrypted data computation but also shed light on practical challenges and considerations when working with homomorphically encrypted data.

Chapter 2: Literature Review: Background of Homomorphic Encryption

2.1 Theoretical Introduction to Privacy Homomorphisms

In 1978, Rivest et al. [1] theoretically studied the concept of what they termed a “privacy homomorphism.” They proposed a scenario: suppose a small loan company would like to securely store their encrypted data on a time-sharing device. The company would also like to be able to answer questions such as

- What is the average loan size?
- What is the expected income from loans in the next month?
- How many loans over some value *v* have been granted?

The small company can consider the following possibilities for answering these questions:

- Reject the idea of the time-shared service, and purchase and in-house system.
- Use the time-sharing service for storing encrypted files only, and have a system in-house for decryption and computation.
- The time-sharing company can use modified hardware which allows brief decryption within the CPU, which is not externally accessible.
- Use a special privacy homomorphism to encrypt data, so that the time-shared computer can operate on the data without decrypting it first.

The first two options don’t allow efficient use of remote storage of data, if that data needs to be computed on. The third option is workable, but requires a special cooperation of the time-sharing company. The fourth option, however, would be ideal, but requires a *privacy homomorphism.* Rivest et al. [1] propose that a privacy homomorphism could theoretically be a solution.

2.1.1 Mathematical Representation of a Homomorphism

A *homomorphism* is a mathematical concept which describes a structure-preserving map between two algebraic structures. A homomorphism can translate one set of operations in one structure to another set of operations in another structure, while maintaining the relationship between elements. Let $A,\ C$ be sets with operations $\circ_A$ and $\circ_C$. The function f is a homomorphism from A to C if for all $x, y \in C$, $f(x \circ_C y) = f(x) \circ_A f(y)$.

Suppose we want to perform some operation + on $x', y' \in A$. Consider a decryption and encryption pair of operations, $\phi^{-1}: A \rightarrow C$ and $\phi: C \rightarrow A$, respectively, and some function $+_C$ such that $\phi: C \rightarrow A$ is a homomorphism. That is, we want

$$\phi(x +_c y) = \phi(x) + \phi(y) = x' + y'.$$

If we identify an encryption and decryption scheme that is homomorphic, it theoretically allows for calculations on encrypted data. This means data sent to a third-party and operations can be performed on it in its encrypted state.

There are some immediate restrictions to such a homomorphism that would prevent it from being cryptographically secure. For example, Rivest et al. [1] point out some inherent restrictions that limit the utility of privacy homomorphisms. For example, if there is a predicate operation "≤" which allows for total order of arbitrary constants, there is no secure privacy homomorphism from C to A.

If there is a predicate operation such as "≤", any encryption d_i can easily be decoded by systematically comparing encryptions of known values to the encryption of d_i, to find where d_i fits into the sequence. That is a malicious user can decode $\phi^{-1}(d_i)$ by computing $\phi^{-1}(1) = 1'$,

$\phi^{-1}(2) = 1' + 1'$, $\phi^{-1}(4) = \phi^{-1}(2) + \phi^{-1}(2)$, and so on, until finding a k such that

$\phi^{-1}(2^k) \geq \phi^{-1}(d_i)$. Using this strategy d_i can be computed exactly [1].

2.1.2. Example of a Theoretical Privacy Homomorphism

Rivest et al. [1] give an example to illustrate that such privacy homomorphisms might exist in theory (though they acknowledged that their examples were weak cryptographically) Consider $A =< Z_{p-1}; +_{p-1}, -_{p-1} >$, the system of integers modulo $p - 1$ with the operations of addition and subtraction, where p is a prime number. We may choose $C =< Z_n; \times_n; \div_n >$, the integers modulo n where $n = pq$, the product and p and a large prime q. Let g be a generator modulo p. Then we choose

$$\phi^{-1}(x) \equiv g^x (mod\, n)$$

and the decoding function is the inverse "mod(p) logarithm, base g" function. By laws of exponents, ϕ is a homomorphism. If n is difficult to factor (both p and q are large) and the prime p is such that logarithms modulo p can be efficiently computed, then the computer system can give both g and n without fear of compromising the security of the data.

While this example, and the others given by Rivest et al. [1] aren't necessarily strong cryptographically, they illustrate that *privacy homomorphisms*, which have turned into what we now call *homomorphic encryption*, are possible in theory. This led to a search for secure, homomorphic encryption schemes.

2.2 Learning with Errors

The Homomorphic Encryption schemes that were to come would be built on the mathematics and security of Learning with Errors (LWE). The LWE problem was introduced by

Oded Regev in 2005 [3], [4], and later extended to Rings by Lyubashevesky, Peikert, and Regev [14].

The security of LWE is based on the hardness of worst-case problems on ideal lattices. This means that if there is an efficient algorithm to solve the average LWE problem, there is also an efficient algorithm to solve the worst-case problem on ideal lattices [3]. Since the hardness of the ideal lattice problem is well documented [4, 15], this gives a strong indication of the hardness of the LWE problem.

We discuss the mathematics of LWE now, and then later introduce the FHE schemes built with LWE.

2.2.1 Mathematical Basis of LWE

LWE is based on a system of linear equations, with an error introduced to prevent the system prom being truly uniform.

Regev presents the cryptosystem parameterized by integers n (the security parameter), m (number of equations), q (modulus), and a real $\alpha > 0$ (noise parameter). A choice that guarantees both security and correctness [3] is as follows. Choose q to be a prime between n^2 and $2n^2$, $m = 1.1 \cdot n \log q$, and $\alpha = 1/(\sqrt{n} \log^2 n)$. Below is a description of the scheme. All additions are performed modulo q.

Private Key: The private key is a vector $\mathbf{s}$ chosen uniformly from $\mathbb{Z}_q^n$.

Public Key: The public key consists of m samples $(\mathbf{a}_i, b_i)_{i=1}^{m}$ form the LWE distribution with secret $\mathbf{s}$, modulus q, and error parameter α.

Encryption: For each bit of the message, do the following. Choose a random set S uniformly among all 2^m subsets of $[m]$. The encryption is $(\Sigma_{i \in S} \mathbf{a_i}, \Sigma_{i \in S} b_i)$ if the bit is 0 and $(\Sigma_{i \in S} \mathbf{a_i}, \lfloor q/2 \rfloor + \Sigma_{i \in S} b_i)$ if the bit is 1.

Decryption: The decryption of a pair $(\mathbf{a}, b)$ is 0 if $b - \langle \mathbf{a}, \mathbf{s} \rangle$ is closer to 0 than to $\lfloor q/2 \rfloor$ modulo q and 1 otherwise.

2.2.2 Example of a Simple LWE System of Linear Equations

We now provide an example to illustrate the mathematics above.

Let $n = 5, 5^2 \leq q = 29 \leq 50$, $m = 1.1 \cdot 5 \log 29$, $\alpha = 1/\left(\sqrt{5} \log^2 5\right)$. Using a uniform distribution from $\mathbb{Z}_{29}^5$, we use the private key $\mathbf{s} = [4,\ 27,\ 12,\ 18,\ 8]$.

We generate a random set of 8 samples from the LWE distribution with secret key $\mathbf{s}$, modulus $q = 29$, and add the error parameter α. We end up with the following in the Public Key set. Note that we have separated the error for the sake of clarity with the example.

$$24s_0 + 22s_1 + 1s_2 + 2s_3 + 11s_4 = 14 + 1 \ (mod\ 29)$$

$$11s_0 + 15s_1 + 5s_2 + 18s_3 + 10s_4 = 14 + 0 \ (mod\ 29)$$

$$11s_0 + 20s_1 + 9s_2 + 25s_3 + 8s_4 = 17 + 2 \ (mod\ 29)$$

$$22s_0 + 1s_1 + 23s_2 + 3s_3 + 23s_4 = 20 + 0 \ (mod\ 29)$$

$$17s_0 + 22s_1 + 10s_2 + 1s_3 + 1s_4 = 25 + 0 \ (mod\ 29)$$

$$6s_0 + 20s_1 + 14s_2 + 0s_3 + 17s_4 = 27 + 0 \ (mod\ 29)$$

$$13s_0 + 12s_1 + 7s_2 + 13s_3 + 10s_4 = 20 + 2 \ (mod\ 29)$$

$$17s_0 + 19s_1 + 12s_2 + 3s_3 + 20s_4 = 11 + 1 \ (mod\ 29)$$

Encryption: For each bit we wish to encrypt, choose a random subset, in this case the elements 0, 1, 2, 3, 6. Add these equations together modulo 29. Note that without errors, the right hand side is 27. The added error of 5 results in the right hand side being 3.

$$23s_0 + 12s_1 + 16s_2 + 3 + 4s_4 = 3\ (mod\ 29)$$

If we wish to send an encrypted 0 bit, we send this equation as-is. If we wish to send an encrypted 1 bit, we need to add $\lfloor 29/2 \rfloor = 14$, and so should instead send

$$23s_0 + 12s_1 + 16s_2 + 3 + 4s_4 = 17\ (mod\ 29)$$

Decryption: The decryption of a pair $(\mathbf{a}, b)$ is 0 if $b - \langle \mathbf{a}, \mathbf{s} \rangle$ is closer to 0 than $\lfloor q/2 \rfloor$.

In the first equation, $b - \langle \mathbf{a}, \mathbf{s} \rangle = 27 - 3 = 24$. This is closer to 0 than to 14, so this bit would decrypt as 0.

In the case of the second equation, $b - \langle \mathbf{a}, \mathbf{s} \rangle = 27 - 17 = 10$. This is closer to 14 than to 0, so decrypts as a 1.

The FHE schemes to come were built on the mathematics and the hardness assumptions of the LWE problem and its variations.

2.3 Somewhat Homomorphic Encryption

Following the introduction of the idea, homomorphic encryption changed from a theoretical idea to implemented schemes. The first schemes developed were Somewhat Homomorphic Encryption (SHE) schemes, meaning they were homomorphic on addition or homomorphic on multiplication, but not on both, or, they were homomorphic for some number of operations, but eventually contained too much noise decrypt accurately.

The first semantically secure homomorphic encryption scheme was proposed by Goldwasser and Micali [16] in 1982. Their scheme is additively homomorphic. In 2006, Boneh, Goh, and Nissim [17] introduced a homomorphic encryption scheme that was homomorphic

under addition, along with *one* multiplication operation. This allows for evaluation of 2-DNF circuits (that is, two or-circuits combined by one and circuit). Their construction allows for unlimited additions, one multiplication, followed by unlimited additions.

The scheme relies on the *subgroup decisions problems*, on a new hardness problem put forward by Bohen et al., in which they prove that given an element of a group of composite order $n = q_1 q_2$, it is infeasible to decide whether it belongs a subgroup of order q_1.

The scheme built by Boneh et al. [17] takes polynomial time in the size of the message space T, so can only be used to encrypt short messages.

2.4 Gentry's Fully Homomorphic Encryption Scheme

In 2008, Gentry pioneered the first Fully Homomorphic Encryption (FHE) Scheme. This means that the scheme allowed for unbounded use of both multiplication and addition, meaning that any function could technically be computed.

The scheme built by Gentry is broken down into three steps: a general "bootstrapping" result, an "initial construction" using ideal lattices, and a technique to "squash the decryption circuit" to permit bootstrapping.

In Gentry's scheme, a ciphertext has the form $v + x$ where v is the ideal lattice and x is an "error" or "offset" vector. The ideal lattice scheme follows from the LWE, as they are isomorphic [3]. On its own, the scheme is only homomorphic for shallow circuits due to the linear growth of the "error" vector with addition and its exponential growth with multiplication. As explained in the LWE section above, the purpose for the "error" vector, or noise, in FHE schemes is because the noise is what guarantees the security of the fresh encryption. To address the issue of noise growth, a bootstrapping step is necessary [5].

2.4.1. Bootstrapping

The idea behind the bootstrapping step is, after some number of operations, to reduce the amount of noise back to the "original" amount to allow for larger circuits. If the noise grows too large, the ciphertext will reach a point where it no longer is able to be decrypted. When the limit is being reached, a bootstrapping step can be performed to reduce the amount of noise.
A scheme is termed "bootstrappable" if it can homomorphically evaluate its own decryption circuit and still handle at least one more operation. That is, the bootstrapping scheme homomorphically decrypts the ciphertext. In normal decryption, the secret key is used to output a plaintext. With bootstrapping, the encrypted secret key is used to output a new encryption, and this new encryption has a smaller "error" vector (or less noise) that the original ciphertext.

The reason that the bootstrapping operation must perform an additional non-trivial operation is because otherwise, the eliminated noise will be canceled out when performing the subsequent operation.

The fact that Gentry's scheme is bootstrappable is what made it a FHE scheme. Unlimited multiplication and addition were theoretically possible. However, the bootstrapping step is expensive, so while Gentry's scheme is quite practical for shallow circuits, due to the computational overhead of the bootstrapping operation, Gentry's scheme becomes less practical for applications requiring numerous multiplications [5].

2.5 Reductions to the Bootstrapping Cost

The central technique of Gentry's scheme was *Bootstrapping*. This is what allowed Gentry to make the breakthrough scheme from somewhat to fully homomorphic encryption. With the introduction of bootstrapping, Gentry found a way to address the growth of noise. The bootstrapping allowed for homomorphically evaluation of the SHE's decrypting function on the

ciphertext with too much noise. This reset the noise, thus allowing for further computation [11]. With Gentry's technique, there were no longer any theoretical limitations to what computations can be performed on the ciphertext. However, the time requirements were the bottleneck preventing the scheme from being feasible in practice [11].

Because of this, reductions to bootstrapping costs have been one of the major areas of research in FHE. [5, 6, 10, 11, 19, 20]. Some of these advancements are described below.

2.5.1 Reducing Bootstrapping Costs with Modulus Switching

Works such as Gentry's addressed the issue of noise by using the bootstrapping step to "squash" the noise [5]. In 2012, Brakerski and Vaikuntanthan introduced a novel technique, *dimension-modulus reduction*. This method reduced the decryption complexity by shortening ciphertexts without necessitating additional assumptions.

While Gentry's construction for SHE was rooted in the complexity of problems on ideal lattices, Brakerski and Vaikuntathan constructed a SHE whose security relied on the hardness of arbitrary lattices, not just ideal lattices [10]. This advancement was built on the LWE problem. Encryptions were represented by linear functions with noise. Though addition was straightforward, multiplication rapidly expanded the size of ciphertexts [10].

To counteract this, Brakerski and Vaikuntanthan introduced "*re-linearization*." This technique allowed multiplied ciphertexts to be expressed with a size roughly the same as the initial ciphertexts. The process entailed creating a chain of encrypted linear and quadratic terms, which, upon re-linearization, generated a function representing the multiplication of the initial two ciphertexts. This re-linearization process used a chain of secret keys. With repeated multiplication, noise growth necessitated a bootstrapping technique to convert the scheme from "somewhat" to "fully" homomorphic [10].

To achieve full homomorphism, Gentry's bootstrapping depended on the hardness of the sparse subset-sum problem. The purpose of the bootstrapping step was to reduce the noise introduced during addition and multiplication. Gentry's scheme used "squashing" to reduce decryption complexity, but at the cost of introducing the sparse subset problem assumption. In contrast, Brakerski and Vaikuntathan's scheme had similar homomorphic capacity, but a more compact decryption circuit. Crucially, their method did not introduce any additional assumptions, relying solely on LWE [10].

They did so by using *dimension-modulus reduction*. The core idea was converting a ciphertext with parameters (n, log q) to another representing the identical message but with altered parameters (k, log p), without compromising the message's integrity. This transformation, akin to the re-linearization process, used a series of public parameters for ciphertext conversion. This new bootstrapping technique relied only on the LWE assumption, improving the scheme's efficiency [10].

2.5.2 Bootstrapping in Quasilinear Time

In 2013, Alperin-Sheriff and Peikert [11] were able to find a faster bootstrapping method with polynomial error. Their bootstrapping algorithm provided methods that were in quasilinear time for both "packed" and "non-packed" ciphertexts. The main technique that they used was to enhance the "ring-switching" procedure of Gentry et al. In their algorithm, they enhance the "ring-switching" procedure to support switching between two rings where either is a subring of the other. This allowed them to provide more efficient homomorphic methods for evaluating many linear transformations, including the decryption function [11].

The algorithm by Alperin-Sheriff and Peikert was algorithmically simpler than previous methods. Their method for non-packed ciphertexts used only cyclotomic rings having

power-of-two index, which allow for a fast implementation [11]. For the packed ciphertext method, their procedure drew on high-level ideas from [7, 8], but the actual implementation was different conceptually. It avoided permutation networks and permutations of plaintext slots, as well as avoided relying on general-purpose compilers for evaluating homomorphic circuits, but instead introduced new procedures for homomorphically mapping between encrypted texts and plaintext slots. [11]

In addition to improving the time of the bootstrapping procedures, their method was entirely algebraic and the full procedure could be described as elementary operations from the native instruction set of the SHE scheme. This simplicity affected the concrete efficiency of the bootstrapping procedure [11]. Their method also decoupled the algebraic structures of the SHE plaintext ring versus the ring needed for bootstrapping [11].

2.5.4 TFHE: Fast Fully Homomorphic Encryption over the Torus

In 2015, Ducas and Micciancio [12] presented a very fast bootstrapping procedure, of about .69 seconds, which was a big step towards practical FHE for arbitrary circuits. They began by analyzing bootstrapping *in vitro*, or in the simplest possible setting: with two encrypted bits $E(b_1)$ and $E(b_2)$, they wanted to obtain the encrypted result $E(b_1 \barwedge b_2)$ in a form similar to the input bits. The encryption they used is a standard lattice encryption scheme, so $E(b_i)$ are noisy encryptions and the output ciphertext $E(b_1 \barwedge b_2)$ is bootstrapped to reduce its noise level. Their new bootstrapping method allowed for performing the computation in less than a second on consumer grade personal computers [12].

Ducas and Micciancio achieve these results based on two main techniques. First, they introduce a novel homomorphic NAND operation. With two encryptions $E(m_1)$ and $E(m_2)$, one

can compute a noisier $E(m_1 + m_1)$. Instead of working in modulo 2, they extend this to arithmetic modulo 4, to achieve a logical NAND operation. The outcome is that $E(m_1 \bar{\wedge} m_1)$ can be obtained with a simple transformation. This new homomorphic NAND operation introduces less noise than previous techniques, simplifying the bootstrapping process [12].

Their second contribution is an enhancement of bootstrapping. Building off of the work from [11], they use a homomorphic cryptosystem that encrypts integers mod q for efficient scalar product calculation. They also introduce a ring variant to the method used by [11]. By directly encoding cyclic groups and using the structure of lattices, they can represent cyclic group elements with just one ciphertext [12].

In 2016, Chillotti, Gama, Georgieva, and Izabachène further improved the bootstrapping procedure. Their FHE scheme involves using polynomials over the real torus, and combines the Scale-Invariant-LWE problem of [20] or the LWE normal form of [21] with the General-LWE problem of [22]. They call their scheme TLWE, and it is a unified representation of LWE ciphertexts, which encode polynomials over the Torus [18]. Their scheme extends the work of [9, 23, 12], and the efficiency comes from combining TLWE and TGSW. This technique was also used independently by [24].

Chillotti et al. expand on the previous work like this. A TGSW sample is essentially a matrix whose individual rows are TLWE samples, and so the external product of TGSW times TLWE is quicker than the internal product TGSW times TGSW used in previous work. This is akin to comparing the speed of computing a matrix-vector product to a matrix-matrix product. As a result, their bootstrapping procedure is 12 times faster than the previously most efficient bootstrapping procedure [12], and runs at less than 0.052s [18].

Chillotti et al. continued to develop their TFHE scheme. In 2017, they released a paper that included techniques for packing several bits of information and using the compact representations to either batch multiple or speed-up single operations. This helped to address one of the drawbacks of FHE schemes, which is the huge expansion factor of ciphertexts to plaintexts. For example, in some cases they reduced the expansion from where it was between 6400 in output and 384000 in input to an expansion of about 64. Using their packing technique, and packing both vertically and horizontally, they were able to reduce bootstrapping to 137*ms* [19].

Over the course of the years since Gentry first introduced his Fully Homomorphic Encryption Scheme in 2008, there has been a lot of work done to reduce the bootstrapping requirements. Many FHE implementations have been built, including HElib [25], Microsoft SEAL[26], TFHE [13], and OpenFHE [27] to name a few. Zama [28, 30] has developed a framework that contains a TFHE Compiler to make the process of writing FHE programs easier for developers. These advances have resulted in FHE schemes that can be implemented on personal devices, and in the next section, implementing Concrete by Zama to build a blind-bidding auction is discussed.

Chapter 3: Blind-Bidding Auction

In a blind bidding auction, each participant in the auction wants to win with the highest bid, but all bids should be kept private through the entire bidding process. The only bid that should be revealed is the winning bid. All information about all other bids should remain private. There may be multiple winning bids which are tied. In this case, all of these bids should be revealed.

We now explain the blind-bidding auction we have developed. In this section, we outline the system requirements, methods, and implementation of our blind-bidding auction.

3.1. System Requirements

The blind-bidding auction is designed with the following system requirements:

- Each bid contains two parts: The bid value and a bidder ID (for identifying the owner of the winning bid).
- The maximum bid(s) are calculated. If there is more than one maximum, then all tied maximums will be output. These maximum bids will be decrypted. Nothing else will be decrypted. The decrypted values reveal both the bid values and the bidder IDs for these bids.
- No information is learned about any other bids.

3.2. Methods

The methodology used is experimental research. We implement the bidding system under various parameter settings and document the performance metrics such as computational time and accuracy under each setting.

We build the blind-bidding auction using Rust, using the Concrete Fully Homomorphic Library by Zama. We use a function from Optalysys [29] for computing the maximum between two encrypted values, which utilizes Concrete's programmable bootstrapping.

3.2.1 Justification for Tool Selection

Concrete by Zama is a Rust-based, open source framework enabling developers to use homomorphic encryption without needing to understand all of the cryptography. The Zama Team released Concrete officially on July 7, 2022. Zama is a company specializing in privacy-preserving technologies. Concrete addresses three of the major issues in FHE: too slow, too hard to use, and too limited in functionality.

There are two main approaches to FHE. The "leveled" approach attempts to only do as many computations as possible before noise overflows into the data. The "bootstrapped" approach adds in a bootstrapping operation to reduce noise (but increase computation time). Concrete implements a variant of TFHE [13] that supports both leveled and fast bootstrapped operations, as well as approximate or exact evaluation of arbitrary functions.

Concrete is the first framework to introduce programmable bootstrapping, a technique where a univariate function can be computed for free during the bootstrapping operation. However, this comes at the tradeoff of small precision, currently limited to 16 bits [28, 30].

3.3 Implementation and Challenges

In this section, we describe the implementation of the blind-bidding auction. One of the challenges of FHE is that the complexity quickly increases as size increases. Because of this, the focus of this study is on implementing the blind-bidding auction for small values, particularly focusing on the needed logic for working with the encrypted data.

While the auction serves as a testbed to better understand the complexity and nuances of FHE, it is not intended to be a production-level implementation. The auction described below does not work for sufficiently large values to make it practical, but does help reveal the many challenges and nuances of FHE. Of particular interest in this section is the algorithm used to compute the maximum bid(s), without revealing any information about any losing bids.

3.3.1 Data Input and Pre-Processing

Each bidder is assigned a bidder ID. Based on the number of bidders, a sufficient number of the least significant bits will be reserved for storing the bidder ID. The remaining bits will store the bid value.

For example, if there are 9 bidders, they will be assigned bidder IDs between 1–9 (nobody will be assigned 0). The right-most digit (base 10) will be used to store the bidder ID. The remaining digits will be used to store the bid value. If the bidder with bidder ID 5 would like to submit a bid of 76, their bid value in plaintext will be 765.

In a production implementation of the blind-bidding auction, implementation would need to be added to allow for the sharing of the public key, and for individual bidders to securely encrypt their bids via the public key. The focus of this study is on the feasibility of using FHE to sort and output only the winning bid(s), after they are encrypted. So, for the sake of this study, bids are all accepted and encrypted via the same program used for sorting the bids. In the next section, we explore the implementation of the bidding logic.

3.3.2 Bidding Logic Implementation

When considering the most efficient way to calculate the winning bid, we need to consider the constraints of FHE. We cannot use any logic such as directly comparing whether $a < b$ for some encrypted a, b. The process of testing for the maximum between two pairs a and

b results in the values c and d, where we know, for example, that $c < d$, but can't determine whether $c = a$ and $d = b$ or $c = b$ and $d = a$.

Due to the specific limitations of FHE, traditional algorithms for finding the maximum values may not be directly applicable. Instead, we utilize a homomorphic `compute_max_min()` function which we describe below. The `compute_max_min()` function comes from a paper by Optalysys [29], a company that is developing a silicon-photonic chip specialized to speed up FHE operations. Following this, we describe our method for finding the maximum bids. A comprehensive analysis of this method's performance can be found in the *Performance Results* section below.

3.3.3.1 Homomorphically computing the maximum and minimum of a pair

Finding the maximum values in the array of bids relies on the `compute_max_min()` function. This function is from the implementation by Optalysys [29]. The computation of the maximum and minimum values happen during a programmable bootstrap operation. We first explain the simple algorithm of computing the maximum and minimum, and then describe the implications of computing this algorithm in the context of homomorphic encryption.

The algorithm's logic is simple. If you have two ciphertexts c_1 and c_2, you can compute the maximum and the minimum as shown in Algorithm 1.

Algorithm 1

Computing the maximum and the minimum values given two encrypted ciphertexts

```
compute_max_min()

1 difference <- cipher2 - cipher1
2 if difference > 0
3    differencePositive <- difference
```

```
4 else
5    differencePositive <- 0
6 end if
7 maximum <- cipher1 + differencePositive
8 minimum <- cipher2 - differencePositive
```

We present a numerical example to demonstrate this algorithm. First for the case of `cipher1>cipher2`.

Let `cipher1=5` and `cipher2=2`. We now compute maximum and minimum based on this algorithm. We compute `difference` as `cipher2 - cipher1 = 2-5 = -3`. Because `difference` is not > 0, we set `differencePositive` to `0`. We then find `maximum` and `minimum` by setting `maximum=5+0=5` and `minimum=2-0=2`.

Now let's do the same thing with the cipher values swapped, where `cipher1=2` and `cipher2=5`. We compute `difference` as `cipher2 - cipher1 = 5-2 = 3`. Because `difference > 0`, we set `differencePositive` = `difference = 3`. We then find `maximum` and `minimum` by setting `maximum=2+3=0` and `minimum=5-3=2`.

As you can see, the correct minimum and maximum values are returned in both the case of `cipher1>cipher2` and `cipher1<cipher2`. Having seen the algorithm in action with a numerical example, let’s describe the underlying reasons for setting it up in such a way.

The reason to set the algorithm up in this fashion is due to the homomorphic encryption and the possibility for the programmable bootstrap step. We need to recall that at each step, when everything is encrypted homomorphically, any operation result is also encrypted. In line 1 of Algorithm 1, the difference that is computed between `cipher2` and `cipher1` is an encrypted value. Thus, we can’t simply use logic at that point to return the maximum and minimum values.

Instead, in the setup of this function, Optalysys utilized programmable bootstrap [29]. In Concrete, there is a function `bootstrap_with_function()`. The function computes a bootstrap and applies an arbitrary function to the LWE ciphertext. In addition to a bootstrapping key, *f,* a function to apply, is given as an argument to `bootstrap_with_function()`. The output `bootstrap_with_function()` is the encrypted evaluation of *f* [29].

It is of importance to us that we can compute a function within the bootstrapping computation. First, `cipher_diff`, the difference between `cipher1` and `cipher2` is computed. The function that we compute within the bootstrapping computation allows us to check if `cipher_diff` ≥ 0. If it is, the function returns a new encryption of `cipher_diff`. If it is not, it returns an encryption of 0. Even though this is encrypted, we now don't need to know what the value of this returned value is, and can still apply it to our algorithm, to compute the maximum and minimum.

As we describe our `identify_max_bids()` function in the next section, it is important to recall that, while `compute_max_min()` returns an encryption of the maximum and the minimum values, these two returned values are indecipherable from the input `cipher1` and `cipher2` values. This inability to know whether the starting encryption `cipher1` or `cipher2` is larger impacts the decisions made in implementing the `identify_max_bids()` algorithm which we describe in the next section.

3.3.2.2 Finding the Maximum Values

We find the maximum values using a modified version of bubble sorting. In a bubble sort, going from left to right, two values are compared to each other. If the value on the left is larger than the one on the right, they are swapped. Then that current right value is compared to the

value on its right. After one iteration through the array, the maximum value is moved in the rightmost position.

If the goal is to use bubble sort to sort an array completely, the process would continue until the 2nd largest value is in the second rightmost position, the 3rd largest value is in the third rightmost position, and so on, until the array is fully sorted. In our case, we don't necessarily need to sort the full array, we only need to find the winning bid, or, the tied winning bids.

For the sake of explanation, let's discuss a hypothetical set of bids, {21, 22, 13, 14}. In this case, there are three bidders. Their IDs are 1, 2, 3, and 4. Bidders with ID's 1 and 2 have bid 2, and bidders with ID's 3 and 4 have bid 1.

After looping through the array fully the first time, it ends up in the arrangement of {21, 13, 14, 22}. At this point the highest bid is guaranteed to be the rightmost position. If there are tie bids (which there is in this case), then the value in the right most position is the tied bid with the highest bidder ID.

Since we know that 22 is a winning bid (because the rightmost bid must be the highest), we can decrypt it. However, we now need a way to determine if there are any tie bids. Remember, we are working with encrypted data. We don't know anything about which bids are 2s and which bids are 1s. The initial thought might be to just sort the array again for the n-1 elements, and check the second highest bid. However, we need to be careful–we don't want to learn any information about losing bids. If we do this second sort, and it turns out we only have one high bid, then we'll learn information that is confidential by decrypting the second largest value.

So, what we do is we reserve the bidder ID of 0 as a marker. We remove the bidder ID by taking the last digit off of 22 and replacing it with 0, giving us 20. We encrypt this 20, and use it as our marker for the maximum bid value.

We do the sort again on the array for the second largest bid. The result is the array {13, 14, 21}. Next, we use the `compute_max_min()` function, which returns the maximum of two encrypted pairs, to compute the maximum of the current highest (21), and our marker value (20). Since 21>20, when we decrypt the result we see 21. This means we're not done. We need to do another loop through the array to sort and check the next smallest value. This time, we are calling `compute_max_min()` on 14 and 20. Since 20>14, we will get 20 when we decrypt the maximum. At this point, we know we have found all maximum values, because whatever the minimum value is, it is less than 20, and therefore not a winning bid.

As the winning bids are found (22, 21), they are added to an array of maximum bids. The final output of this function is the array of the decrypted values of maximum values.

Algorithm 2: `identify_max_bids()`

Given a set of bids encrypted as ciphertexts, return an array with the winning bid(s)

```
Function identify_max_bids()
     Round <- 0
     MaxArray <- Empty List
     Results <- Ciphers // Initialize Results as a copy of Ciphers
     Results <- sort_for_max(Results, Round)
     CurrentMasterMaxValue <- decrypt last value of Results
     Push CurrentMasterMaxValue to MaxArray
     MasterMaxValue <- remove_bidder_id(CurrentMasterMaxValue)
     MasterMaxValueEnc <- encrypt(MasterMaxValue)
     Done <- false

     While not Done && Round < length(Ciphers) do
           Results <- sort_for_max(Results, Round)
           TempMaxDec <-
     compute_max_min(Results[Length-Round-1],MasterMaxValueEnc)
```

```
            if TempMaxDec > MasterMaxValue then
                 Push TempMaxDec to MaxArray
                 Round += 1
            else
                 Done=true
            end if
      end while
      return MaxArray
End Function

Function sort_for_max(Results, Round)
      if length(Ciphers) < 2 OR Round >= length(Ciphers) - 1
              return Ciphers
      end if
      Results <- Ciphers // Initialize Results as a copy of Ciphers
      for i from 0 to (length(Ciphers) - 2 - Round) do
          (C_max, C_min) <- compute_max_min(Results[i],
      Results[i+1], KSK, BSK, Encoder)
          Results[i]   <- C_min
          Results[i+1] <- C_max
      end for
      return Results
End Function
```

3.3.2.3 Computation Complexity of the `identify_max_bids()` Algorithm.

In this section we analyze the computational complexity of the `identify_max_bids()` function, emphasizing its dependence on both the number of total bids and the number of tied maximum bids.

The `identify_max_bids()` function has a computational complexity of $O(n^2)$. The total run time of the function is dependent on both the total length of the array, and the number of tied winning bids. Because the `compute_max_min()` function is computationally heavy, we care about how many times this function is called. Let n be the total size of the array, and i be the total number of tied maximum bids to be found.

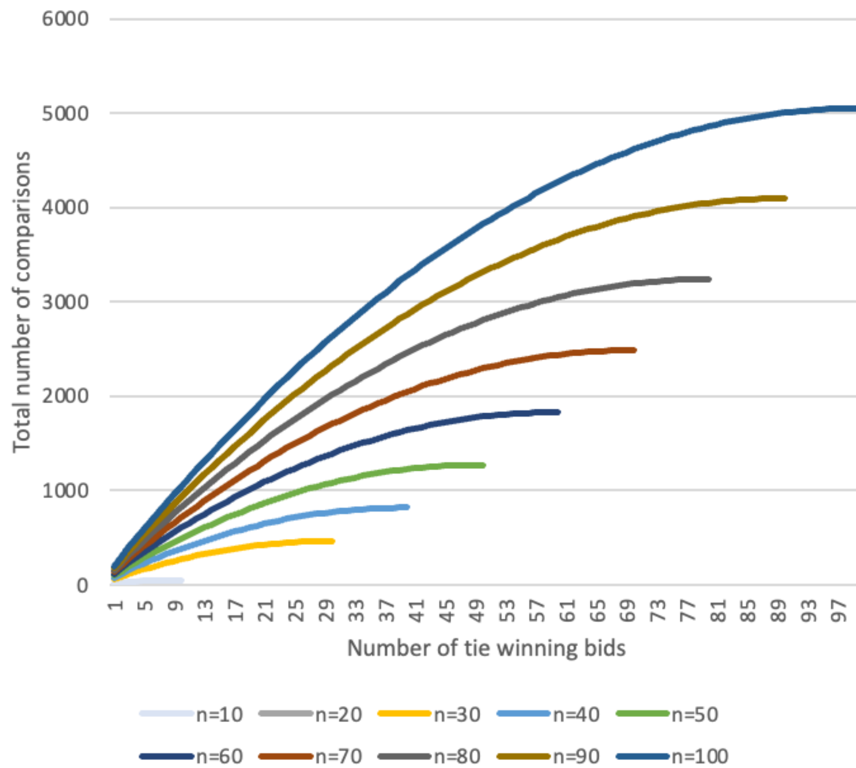

Figure 1. Number of Comparisons in the identify_max_bids() function based on the total number of bids and the number of tie winning bids for arrays of size 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100.

To begin, the algorithm iterates through the array, putting the minimum of each pair on the left, and the maximum on the right. This takes $n - 1$ moves. Each of these comparisons is a call to the `compute_max_min()` function.

The algorithm always iterates through the array a second time, taking $n - 2$ moves in round 2. Then, the value at position $n - 2$ must be compared to the master max value. Each of these comparisons is a call to the `compute_max_min()` function. If this computation shows that the value in position $n - 2$ is less than the master maximum, we are done sorting.

The algorithm then repeats this some number of times, dependent on i. In the end, after the initial sort, the function will be sorted a total of i more times. With each of those sorts, the number of comparisons being made decreases by one.

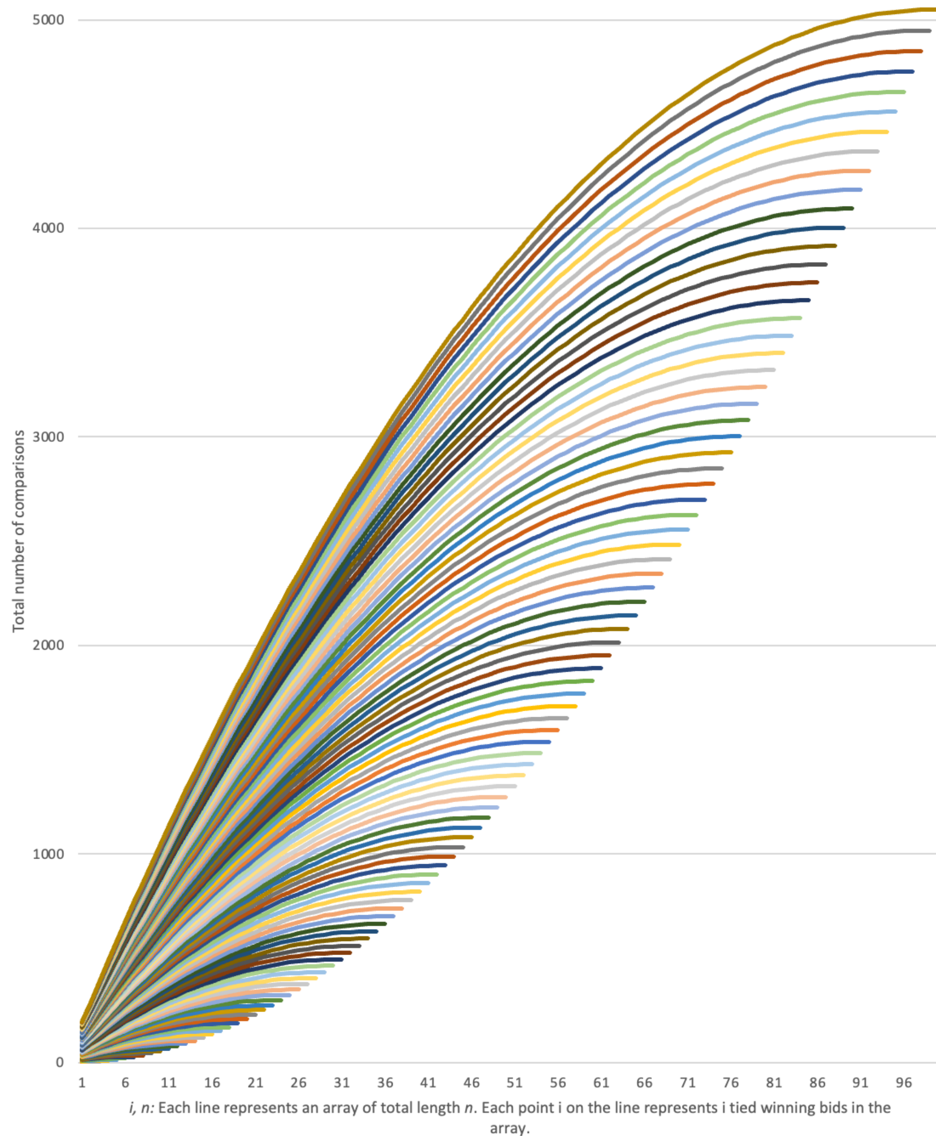

Figure 2. Number of Comparisons in the identify_max_bids() function based on the total number of bids and the number of tie winning bids for all arrays between 1 and 100.

Sorting the array completely takes $\frac{n(n-1)}{2}$ comparisons. After comparison 1, we sort an array whose size decreases by one each time, until we have done so a total of i additional times. This leaves a total of $n - i - 1$ loops not done in the initial n loops expected. The total number of computations to be removed from a complete sort is $\frac{(n-i-1)(n-i-2)}{2}$. After each loop, we also need to compare the current max with the master max value. This comparison happens a total of

i times. Thus, we get the following formula to calculate the total number of times that the `compute_max_min()` function is called:

$$\frac{n(n-1)}{2} - \frac{(n-i-1)(n-i-2)}{2} + i$$

In Figures 1 and 2, we can visualize the number of comparisons for fluctuations in n and i. Each line in the graph represents a distinct n value. Which n is being represented by an individual line can be identified by the x-axis value at the end point of the line. All n values between 1 and 100 are shown in Figure 2. Fewer are shown in Figure 1 for a clearer view of individual lines. For a given i value along the x-axis, you can see on the line representing a given n value, the total number of comparisons. You can see in the chart that as n increases, the total computational complexity increases exponentially. For a given n, as i increases, the amount of additional computation for each additional i is less then for the previous i.

Chapter 4: Results

In this section, we analyze the computational aspects of the FHE scheme and the blind-bidding auction, examining the effects of parameter modifications on computation times and auction accuracy.

4.1 Computation Times

In FHE, computation times can vary significantly depending on the parameters chosen. These are discussed below.

4.1.1 Program Initialization for Different Parameters

There are multiple parameters that need to be set in the Concrete FHE scheme. These include a LWE and RLWE key. We never changed the parameters of the LWE key, always using 128 bits of security, and a polynomial of size 2048. For the RLWE key, we tested for different variations on the security bit and the polynomial size.

We also tested variations with the Key Switching Key (KSK) and Bootstrapping Key (BSK) Initializing the KSK and the BSK both take 2 arguments, base log and levels.

You can view the results for how the initialization speed fluctuates based on changes in the parameters in Table 1.

Using Table 1, we make some observations about the effect of different parameters of the computation time for initiating the KSK and BSK.

Bits of Security: Looking at lines 0 and 5, all parameters stay the same with the exception of the bits of security. Changing from 80 to 128 bits of security had a negligible impact on the time required to initiate the KSK, but increased the time required to initiate the BSK by 28.5%.

Table 1

Computation time required to build the KSK and BSK for different sets of parameters.

	RLWE Bits of Security	RLWE Polynomial Size	(base log, levels)		Computation Time to Initiate	
			KSK	BSK	KSK	BSK
0	80	2048	(4, 5)	(4, 5)	2274 ms	170655 ms
1			(6, 6)	(6,6)	2706 ms	210067 ms
2			(3, 28)	(3, 28)	12599 ms	968697 ms
3	128	1024	(4, 5)	(4, 5)	1188 ms	48371 ms
4		2048			2327 ms	177160 ms
5					2264 ms	238741 ms
6		4096			4604 ms	652111 ms
7					4500 ms	671387 ms
8			(3, 7)	(3, 7)	6480 ms	924656 ms
9			(4, 7)	(4, 7)	6519 ms	970439 ms

Base Log: Looking at lines 8 and 9, all parameters stay the same with the exception of the base log for the KSK and BSK. Increasing the log from 3 to 4 increased the build time of the KSK by 0.6% and the BSK by 4.7%. Compared to some other variations, this parameter's impact is quite negligible on the build time of the KSK and BSK.

Levels: Making a drastic change in the number of levels also results in a drastic change in the initialization time. Jumping the levels from 5 to 28, even while lowering the log base from 4 to 3 increased the initialization time of KSK by 82% and BSK by 82.3%. While this has a big impact in initialization time, such a high level was never actually used past gathering these results, so its impact on accuracy and computation time during the run of the program is unknown.

Changing both Base Log and Levels: Looking at lines 0 and 1, the increasing the values of both the log base and levels increased the KSK initialization time by 16% and the BSK by 18.8%.

Changing the Polynomial Size of the RLWE: Looking at lines 3, 5, and 7, we can see the impact in computation time from changing the polynomial size from 1024 to 2048 to 4096. When changing from 1024 to 2048, it increases the initialization time of the KSK by 47.5% and the BSK by 79.7%. When changing from 2048 to 4069, it increases the initialization time of KSK by 49.7% and of BSK by 64.4%. Overall, to go from a polynomial size of 1024 up to 4096, it increases the computation time of the KSK by 73.6% and the BSK by 92.8%.

Making changes to the polynomial size of the RLWE by far has the greatest overall impact on a whole. It increases initialization time, and, as is shown in later sections, also increases the overall computation time quite significantly.

4.1.2 Influence of Parameters on `compute_max_min()`

Of greater significance than startup speed is the speed of our `compute_max_min()` function. This function is essential for comparing any two encrypted values. Thus, the time to compute maximum values is directly related to the time to run `compute_max_min()`. In Table 2 is a list of the average run time of `compute_max_min()` for a variety of parameters.

Table 2

Computation time required to perform the compute_max_min() function for different sets of parameters.

<table>
<tr><th rowspan="2">RLWE Polynomial Size</th><th rowspan="2">Encoder</th><th colspan="2">(base log, levels)</th><th rowspan="2">Computation of <code>compute_max_min()</code></th></tr>
<tr><th>KSK</th><th>BSK</th></tr>
<tr><td>1024</td><td>(0.0, 1.0, 4, 2)</td><td>(4, 5)</td><td>(4, 5)</td><td>564 ms</td></tr>
<tr><td>2048</td><td>(0.0, 16.0, 5, 2)</td><td>(4, 5)</td><td>(4, 5)</td><td>1116 ms</td></tr>
<tr><td rowspan="5">4096</td><td>(0.0, 31.0, 6, 2)</td><td>(4, 5)</td><td>(4, 5)</td><td>2459 ms</td></tr>
<tr><td>(0.0, 50.0, 6, 2)</td><td>(4, 5)</td><td>(4, 5)</td><td>2450 ms</td></tr>
<tr><td rowspan="3">(0.0, 63.0, 6, 2)</td><td>(4, 5)</td><td>(4, 5)</td><td>2459 ms</td></tr>
<tr><td>(3,7)</td><td>(3,7)</td><td>2968 ms</td></tr>
<tr><td>(4, 7)</td><td>(4, 7)</td><td>3202 ms</td></tr>
</table>

You can see from these results that, on its own, the RLWE Polynomial has a big impact on the computation time of `compute_max_min()`. However, the other parameters also impact the computation time. Making adjustments to the size of values that the encoder handles and to the levels and log base also affect the computation time.

From our exploration of different RLWE polynomial values, it is evident that increasing the size of the polynomial has significant impacts on the computation time. Regardless, to accurately compute large values, it is necessary to do so.

4.1.3 Average vs. Theoretical Computation Time of `identify_max_bids()`

The following section contains an analysis of computation time of `identify_max_bids()`, which relies on the computation time of the `compute_max_min()` function analyzed in the previous section. In the following analysis,

these are the parameters at which the program was run. RLWE polynomial size: 4096; Encoder: min: 0.0, max: 31.0, bits: 6, padding: 2; KSK: base log: 4, levels:5, BSK: base log: 4, levels: 5.

With these parameters, the average time to perform the `compute_max_min()` function was 2305.857 ms. This number comes from the average of 77 calls of `compute_max_mix()`. The timing is calculated using Rust `Instant::now()` directly before and after calling `comput_max_min()`.

The time required to calculate the array of maximum bids was tested, and those results are compared to the theoretical time expected to perform the calculation. As mentioned in section 3.3.2.3, the number of total comparisons in the `identify_max_bids` function is dependent on *n* (the size of the array) and *i* (the total number of tie bids). That equation is

$$\frac{n(n-1)}{2} - \frac{(n-i-1)(n-i-2)}{2} + i$$

Based on this formula and the average computation time of `compute_max_min()`, estimated run times for `identify_max_bids()` are calculated and compared to actual run times in Table 3. All averages in the table come from a sample size of 5.

There were notable outliers in expected vs. actual computation times in the cases when $n = 50,\ i = 40$ and $n = 50,\ i = 50$. Including the major outliers in these two cases, the mean average percentage error (MAPE) is 7.86%. Removing the outlier data points and recalculating the MAPE gives a value of 2.26%.

In the intended behavior of the `identify_max_bids()` function, the maximum value from the first round of sorting is set as the master maximum value. However, discrepancies arose in the aforementioned cases due to a fluctuation in the accuracy during computation. As computations are performed on encrypted values, the associated noise of these values increases.

Table 3

Expected vs. Actual Computation Times for Given n and i Values.

n	i	Comparisons	Expected compute time	Average Actual compute time	Percentage error
9	1	16	36894 ms	37225 ms	0.90%
	2	23	36894 ms	53950 ms	1.73%
	3	29	53035 ms	66690 ms	0.27%
	4	34	66870 ms	77673 ms	0.93%
	5	38	78399 ms	85534 ms	2.38%
	6	41	87623 ms	91301 ms	3.43%
	7	43	94540 ms	95950 ms	3.23%
	8	44	99152 ms	98946 ms	2.48%
	9	44	101458 ms	98737 ms	2.68%
50	5	284	101458 ms	635354 ms	2.98%
	15	679	654863 ms	1543963 ms	1.39%
	25	974	1565677 ms	2146301 ms	4.43%
	40	1229	2245905 ms	1471268 ms	*48.08%
	50	1274	2833898 ms	1864774 ms	*36.52%
	40	1229	2245905 ms	**2718565 ms	4.07%
	50	1274	2833898 ms	**2958624 ms	0.71%

** These two values are outliers. The explanation of the outliers is below*

*** Recalculation of the average, removing outliers*

After 49 comparisons made in the first round of sorting for $n = 50$, the cumulative noise caused the true maximum value of 29 to be mistakenly decrypted as 30. Given that all other tie bids were valued at 2, they became lesser than the master maximum value, which was now

inaccurately set at 30 (that is, a bid value of 3). As a consequence, the `identify_max_bids()` function concluded after just one loop iteration, as opposed to the expected 40 and 49 iterations, respectively. This premature termination resulted in the actual computation time being significantly shorter than the projected computation time. Such discrepancies underscore the challenges of maintaining accuracy in computations involving encrypted values.

4.2 Accuracy

The accuracy of the blind-bidding auction depends on a combination of parameters and configurations. Some of these can be influenced by the auction operator. In this section, we explore how different bidding configurations can affect accuracy.

4.2.1 Fluctuation in Accuracy with Changes in n and i

We focus on the variations in accuracy stemming from variations in n (the total number of bidders), and i (the number of tied maximum bids). Selected results are shown in Table 5. It is worth noting, due to currency parameter constraint, the auction can only accommodate a maximum of 9 unique bidder IDs.

The accuracy tends to decrease as i increases for a given n, reflecting the fact that more comparisons–and thus more opportunities for noise growth–are required for higher i values. As shown in Figure 3, the relationship between accuracy and the size of i has an R^2 value of .6592. This suggests that around 66% of the variation in accuracy is attributed to the increase in i and its associated increase in the number of comparisons. To better understand the sources of the remaining variation, we examined not just the total number of comparisons needed to determine the maximum bids, but also the distribution of how many times each individual bid is being compared.

Table 4

Accuracy of Results of Winning Bids, for Given n and i Values

n	i	**Accuracy**
9	1	100%
9	2	100%
9	3	80%
9	4	79%
9	5	84%
9	6	73%
9	7	74%
9	8	79%
9	9	73%

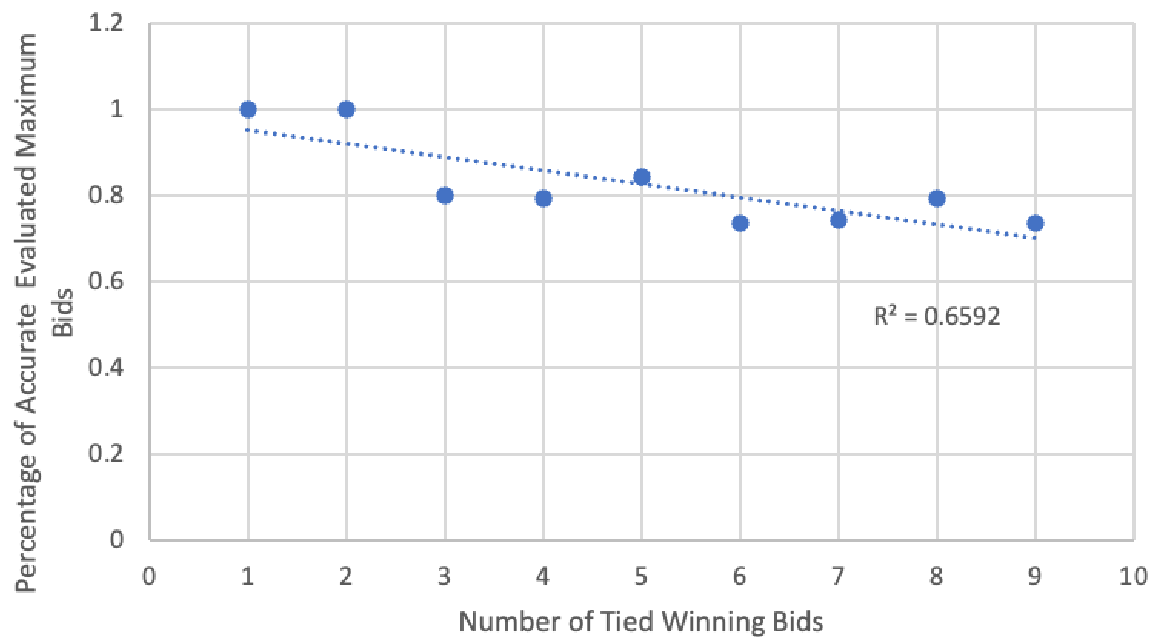

Figure 3. For n=9, Accuracy of Results for i values between 1 and 9.

4.2.2 Fluctuation in Accuracy with Changes in Initial Configuration of Bids

In the auction, each participant is assigned a bidder ID. The bidder ID is not secret, and the initial positioning of bids in the array can be facilitated based on the values of the bidder IDs–either in ascending or descending order. As highlighted in section 3.3.2.2, each bid undergoes at least one comparison with another bid in the process of determining the maximum. Such comparisons result in the re-encryption of the bid, rendering it unrecognizable from its original form. Therefore, the order in which the bids are initially placed in the array, based on bidder IDs, does not compromise the system's security. Moreover, publicly associating original encrypted bids with specific participants doesn't leak any sensitive information.

This section contains an analysis of how the starting configuration of the bids impacts the accuracy of the output. Depending on the initial location of a bid, it may be compared more or fewer times before ending up in its final position, where it is no longer being compared to other bids. We look further into this for sets of 8 bids. We set up three distinct arrays, with bids from the set $B = \{21, 22, 23, 24, 25, 26, 27, 28\}$. We then arranged these in three different way in arrays: With bidder ID in ascending order where $B_a = [21\ 22\ 23\ 24\ 25\ 26\ 27\ 28]$, descending order where $B_d = [28\ 27\ 26\ 25\ 24\ 23\ 22\ 21]$, and randomly where $B_r = [25\ 23\ 21\ 28\ 22\ 27\ 24\ 26]$. We checked the provided maximum bids for ten iterations of the program for each of these configurations. For B_a the accuracy was .87, for B_d the accuracy was .93, and for B_r the accuracy was .9. As you can see, the accuracy is highest when the bids started in the configuration B_d, and this configuration was 6% more accurate than starting configuration B_a.

For each of these variations in starting configuration, there were an equivalent total number of comparisons needed to find the maximum bids. However, the number of times each given encrypted bid was compared to another bid is different for the different configurations. In Table 5 we show the number of times that each encrypted value was compared to the different arrangements. σ is the Standard Deviation of the number of comparisons.

Table 5

Distribution of Comparisons Among Encrypted Values for Different Starting Configurations

Starting Configuration	Total Number of Times the Given Encrypted Value was Compared								
	28	**27**	**26**	**25**	**24**	**23**	**22**	**21**	σ
B_a	1	4	6	8	10	12	14	8	4.224
B_d	7	8	8	8	8	8	8	8	0.354
B_r	5	5	4	11	7	12	11	8	3.137

Considering the fluctuation in the number of comparisons per value in B_a and B_r, we also look at accuracy of, based on the number of times the bid was compared, how likely it is to accurately hold its correct value. We show these results in Figure 4. The relationship between the number of times that a value is compared and the accuracy on decryption (how often it decrypts correctly) has an R^2 value of .7644, suggesting that around 76% of the variation in accuracy is attributed to how often the value was compared. This data suggests that as the number of comparisons increases, the accuracy of a bid decreases. Thus, finding ways to distribute the comparisons among all values will improve overall accuracy. This is discussed further in the discussion section below.

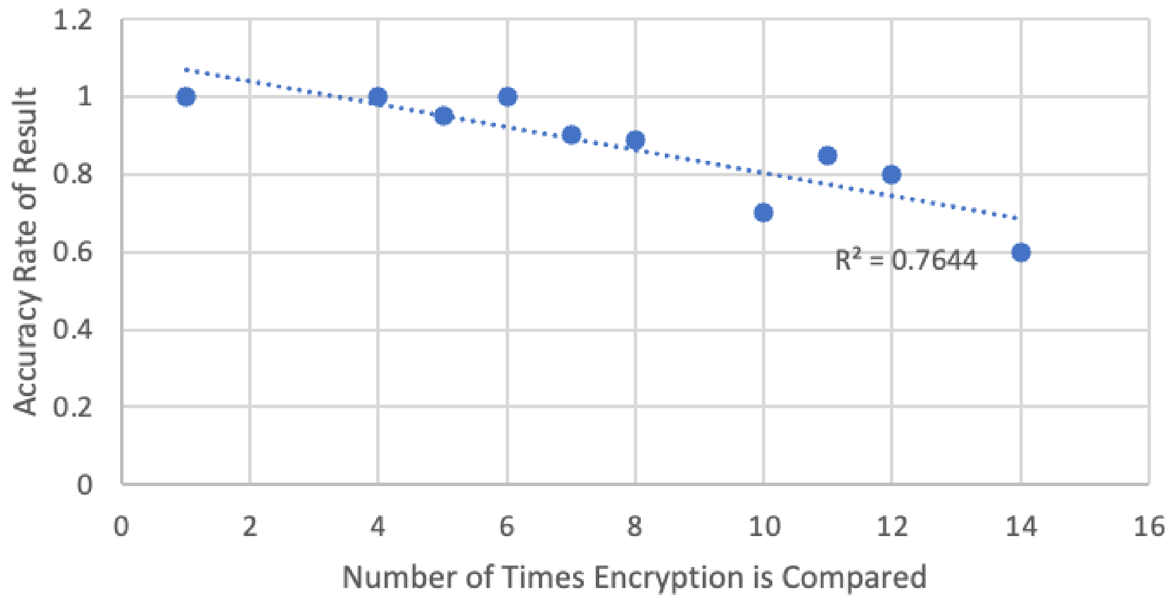

Figure 4. Accuracy of Results Based on the Number of Times Compared

The results underscore the complexities inherent in using a FHE scheme. Operations are notably longer, and accumulation of noise significantly impacts the final accuracy of computations. In the ensuing discussion we consider these accuracy nuances and consider the practicality of the blind-bidding auction.

Chapter 5: Discussion

5.2 Parameter Selection & Accuracy

In a FHE scheme, the potential growth of the error should be tracked well enough that you never end up with inaccurate data, and all noise is taken care of during bootstrapping. Concrete is both a FHE scheme and a Leveled Homomorphic Encryption scheme. With a leveled scheme, there are a set number of levels before each bootstrapping step. With perfect starting configuration of parameters, the auction should have been accurate each time. However, we were not able to achieve this level of accuracy. FHE schemes are complicated, and changes to each parameter impact other parameters.

Due to challenges with maintaining accuracy, interesting patterns show up with regards to how the array is initially configured. This was briefly touched upon in the results in 4.3.2, but we expand on this in the next section.

5.2.1 Accuracy Based on Starting Configuration

As seen in 4.3.2, there is a strong correlation between the number of times an encrypted value is compared and the accuracy of the result. Encrypted values that were compared more times are less likely to still be accurate upon decryption. This raises the question, is there anything the auction owner can do with bidder IDs to optimize the distribution of the comparisons?

As the number of comparisons increases for each individual value, the accuracy decreases. We will delve deeper into how this manifests itself for three unique configurations of the same data. Recall above that the total number of times that `compute_max_min()` is called is

$$\frac{n(n-1)}{2} - \frac{(n-i-1)(n-i-2)}{2} + i.$$

For the majority of these calls, the two values being compared are two full bids. However, for each loop within `identify_max_bids()`, the current max is being compared to the new encryption of the master max value, minus its bid ID. In our equation above, the i was added to the end accounts for this comparison. So, if we let

$$C = \frac{n(n-1)}{2} - \frac{(n-i-1)(n-i-2)}{2},$$

we can compute the sum of the total number of comparison for each individual bid using the equation

$$C * 2 + i.$$

However, if $i = n$, the total number of comparisons in equivalent to when $i = n - 1$. To account for this difference, we calculate the sum of all comparisons as

$$C * 2 - i \text{ , for } i < n$$

$$C * 2 - (i - 1) \text{ , for } i = n$$

Now we further analyze the case of different starting orientations, whose results we gave in 4.3.2. Let a set of bids be $B = \{21, 22, 23, 24, 25, 26, 27, 28\}$. This set contains bids from bidders with ID's 1-8. Consider three variations B_a, B_d, and B_r. In B_a, the bids are entered with bidder IDs in numerical order. In B_b bids are entered with bidder IDs in reverse numerical order. In B_r bids are entered in a random order. We discuss how these configurations affect the accuracy of the sorted outcome.

Let $B_a = [21\ 22\ 23\ 24\ 25\ 26\ 27\ 28]$. We calculate how many times each of these bids is compared to another bid.

First, 21 is compared to 22. They don't swap. 22 is compared to 23. They don't swap. This process continues until 27 is compared to 28.

At this point, 28 is added to the array of winning bids, and it is replaced with an encryption of 20. This is the winning bid (28) without its bidder ID.

During the next round, the same process happens, but the comparisons only go up to 27. Then, the highest bid (27) is compared to the encrypted 20. Since 27 is bigger, it is added to the array of winning bids. Then, the loop runs again.

As you can see, in the first round, the first and last values are compared once and all others twice. This is the case for each round, but since the lower values are still in the array for more rounds, they are being compared for more rounds. In the end, you end up with this distribution of comparisons:

Table 6

Number of Comparisons for Each Value, with an Ascending Starting Configuration

	Total Number of Times Encrypted Value was Compared							
Starting Position	21	22	23	24	25	26	27	28
Number of times compared	8	14	12	10	8	6	4	1

Now we compare the ascending configuration to the descending configuration. Let $B_d = [28\ 27\ 26\ 25\ 24\ 23\ 22\ 21]$.

In the first round, 28 is compared to 27, and they are swapped. Then 28 is compared to 26 and they are swapped. 28 continues to be swapped, and thus continues to be compared to each value until it ends up all the way on the right. This is a total of 7 comparisons.

Just like last time, at this point 28 is added to the array of winning bids, and it is replaced with an encryption of 20. This is the winning bid (28) without its bidder ID.

In round two, 27 is compared to 26 and they are swapped. Then it is compared to 25, and they are swapped. This continues until the array is in this configuration:

26 25 24 23 22 21 27 20

So far, 27 has been compared 7 times, once with the 28 to begin with, and now with each other bid. It gets compared one more time, with 20, to determine if it is one of the maximums. It is, so it gets added to the array of max bids, and the loop continues.

If we track how many times 26 is compared, it is also 8 total. As with all the other values, until 21. Using the starting configuration of bids in descending order, we get the distribution of comparisons shown in Table 7.

Table 7

Number of Comparisons for Each Value, with an Descending Starting Configuration

	Total Number of Times Encrypted Value was Compared							
Starting Position	28	27	26	25	24	23	22	21
Number of times compared	7	8	8	8	8	8	8	8

We can see that using the descending configuration gives a much more uniform configuration. In this case, where everyone has bid the same value, our results will be more accurate if the starting configuration is with bidder IDs in descending order.

However, is this still the case if we go to the opposite extreme, where we only have one winning bid? First, does the starting configuration have the same impact, where bids in descending order lead to a more even distribution of comparisons? Second, what about the winning bid specifically, is it better for that bid to be in ascending or descending order? In Table 8 we show the distribution of comparisons for some different sets of bids, in ascending versus descending bidder ID starting configuration.

Table 8

Initial Configuration of Bids and the Distribution of Comparisons

Starting Configuration									σ
Ascending	11	12	13	14	15	16	17	**28**	1.785
	3	6	6	7	5	5	5	1	
Descending	**28**	17	16	15	14	13	12	11	2.395
	7	8	2	2	2	2	2	2	
Ascending	**21**	12	13	14	15	16	17	18	1.714
	9	3	5	5	4	4	4	4	
Descending	18	17	16	15	14	13	12	**21**	2.736
	9	7	2	2	2	2	2	1	
Ascending	**21**	**22**	13	14	15	**26**	17	18	2.027
	9	9	4	6	6	3	5	5	
Descending	18	17	26	15	14	13	22	21	2.368
	11	8	6	6	4	4	4	4	
Ascending	21	22	13	**24**	15	**26**	17	18	2.222
	10	10	5	7	7	3	6	6	
Descending	18	17	26	15	24	13	22	21	2.332
	12	9	6	6	6	5	5	5	
Ascending	21	22	13	24	15	26	17	28	3.120
	11	11	6	8	9	5	8	1	
Descending	28	17	26	15	24	13	22	21	1.576
	7	11	8	8	7	6	6	6	

As we begin to observe for a larger variety of winning values, it seems that the strategy of placing bids in decreasing bidder ID order does not remain the most uniform distribution when there are fewer winning bids. Based on these examples, it appears that for fewer winning bids, bidder IDs in increasing order leaves a more uniform distribution.

Based on this information, we are unable to conclusively say if there is an advantage on average for starting with bids configured in the ascending bidder ID or descending bidder ID order. This is left as an extension to this work. From the data we have collected, our initial thought is that using the descending bidder ID starting configuration is more valuable, because it leads to a more uniform distribution in the case where there are many ties for maximum. The case with many ties for maximum is also the case with the most comparisons in total, and therefore, optimizing that extreme over the extreme of only one winning bid will likely lead to better results overall.

5.2.2 Speed

For the cases in which we produce accurate data, the speeds are high, but not prohibitive. For example, running the blind-bidding auction for a set of 50 bids, where there are 5 tie bids took an average of 635354 ms to run. Computing the same thing on unencrypted data in Python took .207 ms. The markup in speed is of the magnitude of 3 million times longer. So, while the speed is not in fact prohibitive for a very small use case, as the number of bids increases, and the computation time increases exponentially, that markup in speed very quickly can become prohibitive.

However, as discussed in section 2.5, there have been enormous improvements to FHE computation in the past 15 years, and it continues to be an area of research. Concrete, the FHE implementation used for this project, was just released in 2022 [28, 30]. There continues to be

work on increasing speeds of FHE, such as the work by Optasysis. The `compute_max_min()` function used in this project is from a paper by Optalysys [29]. In that paper, they introduce work being done by their company to develop a silicon-photonic chip to compute the Fourire transform, a main bottleneck in FHE operation, more efficiently. As indicated by their optical simulator, sorting an array which took more than 6s to compute electronically would take .05 s on their silicone-optical chip.

5.3 Practicality for Use

Based on the current status of this program, the program is not practical for actual implementation. The maximum value for which some level of accuracy in results can be achieved is when input is between 0-32, with 6 bits of information. Even then, the smallest bit cannot be used while still achieving accurate results. That means you have 5 bits to store both the bid and bidder information. Leaving at least 1 bit for the bid, you can have up to 16 bidders. Using up to 4 bits for bid leaves you with only 1 bit for storing the bidder. These sizes aren’t practical for actual implementation of the blind-bidding auction.

Chapter 6: Conclusion

6.1 Deliverables

6.2 Deliverables

There are many future extensions to this work. As discussed in the analysis of the accuracy, as the number of comparisons for one particular encrypted value increases, its accuracy decreases. One notable future extension to this work is to analyze further if there are any patterns for starting configurations of the bidder ID’s that can lead to less computations per winning bid. This work analyzes the use case of having multiple tied maximum bids. In such a case, using the initial bid configuration of bidder IDs being in descending order is optimal. However, finding the optimal solution for any ratio of tied maximum bids and total bidders is left to future work.

Other future directions for this work are to implement the same algorithms using other FHE schemes such as HElib [25], Microsoft SEAL [26], or OpenFHE [27], or exploring different algorithms for the sorting of the highest bid.

While the current implementation of the blind-bidding auction using FHE has its challenges, it serves as a foundation to realizing secure and blind auctions in the digital age. Our exploration into the intricacies of FHE, particularly in the context of sorting algorithms, has unveiled both the potential and the hurdles in implementing a blind-bidding auction with FHE. As research of Fully Homomorphic Encryption continues to evolve, we are optimistic that the issues of speed, accuracy, and computational intensity will continue to be improved. This road ahead of Fully Homomorphic Encryption is promising and filled with potential.

References

[1] R. L. Rivest, L. Adleman, and M. L. Dertouzos, “On data banks and privacy homomorphisms,” in *Foundations of Secure Computation, Academia Press*, pp. 169–179, 1978.

[2] C. Gentry and D. Boneh, “A fully homomorphic encryption scheme,” Stanford University, 2009.

[3] O. Regev, "The Learning with Errors Problem," in *IACR Cryptology ePrint Archive*, 2009. [Online]. Available: https://cims.nyu.edu/~regev/papers/lwesurvey.pdf. [Accessed: Oct. 14, 2023].

[4] O. Regev, "On Lattices, Learning with Errors, Random Linear Codes, and Cryptography," in *IACR Cryptology ePrint Archive,* 2009. [Online]. Available: https://cims.nyu.edu/~regev/papers/qcrypto.pdf. [Accessed: Oct. 14, 2023].

[5] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pp. 169-178, 2009.

[6] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, “Fully homomorphic encryption over the integers,” in *IACR Cryptology ePrint Archive*, p. 616, 2009.

[7] C. Gentry, S. Halevi, and N. P. Smart, "Better bootstrapping in fully homomorphic encryption," in *Proceedings of Public Key Cryptography*, pp. 1–16, 2012.

[8] C. Gentry, S. Halevi, and N. P. Smart, "Fully homomorphic encryption with polylog overhead," in *Proceedings of EUROCRYPT*, pp. 465–482, 2012.

[9] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Crypto’13*, 2013.

[10] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) LWE," in *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pp. 97-106, 2011.

[11] J. Alperin-Sheriff and C. Peikert, "Faster bootstrapping with polynomial error," in *Annual Cryptology Conference*, Springer, pp. 297-314, 2013.

[12] L. Ducas and D. Micciancio, "FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second," in *Advances in Cryptology -- EUROCRYPT 2015*, E. Oswald and M. Fischlin, Eds., vol. 9056, Lecture Notes in Computer Science. Berlin, 2015. https://doi.org/10.1007/978-3-662-46800-5_24

[13] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast Fully Homomorphic Encryption over the Torus," GitHub. [Online]. Available: https://github.com/tfhe/tfhe. [Accessed: Oct. 14, 2023].

[14] V. Lyubashevsky, C. Peikert, and O. Regev, "On Ideal Lattices and Learning with Errors over Rings," in *Advances in Cryptology – EUROCRYPT 2010*, H. Gilbert, Eds., Springer, 2010, vol. 6110, doi: 10.1007/978-3-642-13190-5_1.

[15] C. Peikert, "Public-key cryptosystems from the worst-case shortest vector problem," in *Proceedings of the 41st ACM Symposium on Theory of Computing (STOC)*, 2009, pp. 333-342.

[16] S. Goldwasser, S. Micali, “Probabilistic encryption & how to play mental poker keeping secret all partial information,” in P*roceedings of the Fourteenth Annual ACM Symposium on Theory of Computing,* pp. 365–377, 1982.

[17] D. Boneh, E.-J. Goh, and K. Nissim, “Evaluating 2-DNF formulas on Ciphertexts,” in *Theory of Cryptography*, pp. 325–341, 2005.

[18] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *International Conference on the Theory and Application of Cryptology and Information Security*, Springer, pp. 3-33, 2016.

[19] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE," in *International Conference on the Theory and Application of Cryptology and Information Security*, Springer, Cham, pp. 377-408, 2017.

[20] J. H. Cheon and D. Stehlé, "Fully homomorphic encryption over the integers revisited," in *Advances in Cryptology–EUROCRYPT 2015*. Springer. 2015, pp. 513–536.

[21] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "A homomorphic LWE based e-voting scheme," in *Post-Quantum Cryptography*. Springer. 2016, pp. 245–265.

[22] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in *ITCS*, 2012, pp. 309–325.

[23] F. Benhamouda, T. Lepoint, C. Mathieu, and H. Zhou, "Optimization of bootstrapping in circuits," in *ACM-SIAM*, 2017, pp. 2423–2433.

[24] Z. Brakerski and R. Perlman, "Lattice-based fully dynamic multi-key FHE with short ciphertexts," in *Advances in Cryptology - CRYPTO 2016* - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I. 2016, pp. 190–213.

[25] S. Halevi and V. Shoup, "HElib," GitHub. [Online]. Available: https://github.com/homenc/HElib. [Accessed: Oct. 14, 2023].

[26] Microsoft Research, "Microsoft SEAL," GitHub. [Online]. Available: https://github.com/microsoft/SEAL. [Accessed: Oct. 14, 2023].

[27] OpenFHE, "OpenFHE - Open-Source Fully Homomorphic Encryption Library," GitHub. [Online]. Available: https://github.com/openfheorg/openfhe-development. [Accessed: Oct. 14, 2023].

[28] Zama, "What is Concrete?" Zama.ai, Oct. 12, 2023. [Online]. Available: https://docs.zama.ai/concrete. [Accessed: Oct. 14, 2023].

[29] F. Michel, J. Wilson, and E. Cottle, "Max, min, and sort functions using Programmable Bootstrapping in Concrete FHE," Optalysys, Mar. 24, 2022. [Online]. Available: https://medium.com/optalysys/max-min-and-sort-functions-using-programmable-bootstrapping-in-concrete-fhe-ac4d9378f17d. [Accessed: Oct. 14, 2023].

[30] Zama Team, "Introducing the Concrete Framework," Zama, July 7, 2022. [Online]. Available: https://www.zama.ai/post/introducing-the-concrete-framework. [Accessed: Oct. 10, 2023].

Appendices

Appendix A: System Setting, Build, and Compilation

This code was built, compiled, and run on a 2020 M1 MacBook Pro with an Apple M1 chip and 8GB memory. The code was built using the command `cargo build --release --target x86_64-apple-darwin` and run with the command `cargo run --release --target x86_64-apple-darwin`

Appendix B: Cargo.toml

```
[package]
name = "homomorphic_min_max"
version = "0.1.0"
edition = "2021"

[dependencies]
concrete = "0.1.11"
bincode = "1.3.3"
rand = "0.8"
concrete-csprng = "0.3.0"
serde = { version = "1.0", features = ["derive"] }
concrete-core = "^1.0.2"
float-cmp = "0.8.0"
rug = "1.10.0"
```

Appendix C: main.rs

```
use homomorphic_min_max::*;
use std::io;
use std::fs;
mod create_bid;
use std::fs::File;
use std::io::BufReader;
//use std::io::Read;
use bincode;
use float_cmp::approx_eq;
use std::time::Instant;

// Parts of main() are borrowed from [29]
fn main() -> Result<(), Box<dyn std::error::Error>> {

       use crate::create_bid::public_bid;

       // START BORROWED CODE [29]

       let sk_rlwe = RLWESecretKey::new(&RLWE128_1024_1);
       let sk_in = LWESecretKey::new(&LWE128_1024);
       let sk_out = sk_rlwe.to_lwe_secret_key();
       let encoder: Encoder = Encoder::new(0.0, 30.0, 6, 2)?;

       // key switching key
       // Set timer
       let start_time_ksk = Instant::now();
       let ksk = LWEKSK::new(&sk_out, &sk_in, 4, 6);
       let end_time_ksk = Instant::now();
       let elapsed_time_ksk = end_time_ksk.duration_since(start_time_ksk);
       println!("Elapsed time KSK: {} ms", elapsed_time_ksk.as_millis());
```

```
        // bootstrapping key
        // Set timer
        let start_time_bsk = Instant::now();
        let bsk = LWEBSK::new(&sk_in, &sk_rlwe, 4, 6);
        let end_time_bsk = Instant::now();
        let elapsed_time_bsk = end_time_bsk.duration_since(start_time_bsk);
        println!("Elapsed time BSK: {} ms", elapsed_time_bsk.as_millis());

        // END BORROWED CODE [29]

        // Create empty ciphers vector
        let mut ciphers: Vec<LWE> = Vec::new();

        // Initialize number of bids to 0
        let mut num_bids: usize = 0;

        // Ask for the nuber of bids
        println!("Enter the number of bidders: ");
        let mut input_line = String::new();
        io::stdin()
        .read_line(&mut input_line)
        .expect("Failed to read line");
        let num_bids_i32: i32 = input_line.trim().parse().expect("Input not an
integer");

        if num_bids_i32  >= 0 {
        num_bids = num_bids_i32 as usize;
        } else {
        // Handle the case where the index is negative
        eprintln!("Negative index is not allowed.");
        }

        // CREATE THE BID ARRAY
        let file_path = "bid.bin";

        // Check if file exists
        if fs::metadata(file_path).is_ok() {
        // If file exists, delete it
        if let Err(e) = fs::remove_file(file_path) {
               eprintln!("Failed to delete {}: {}", file_path, e);
        }
        }

        // CALL public_bid() one time for each bid
        for _b in 0..num_bids {
        let _ = public_bid(&sk_in, &encoder);
        }

        println!("Done collecting bids: ");

        //READ THE ENCRYPTED MESSAGE FROM CREAT_BID
        let file_path = "bid.bin";

        // Try to open the file
        let file = File::open(file_path)?;

        // Deserialize the content
        let mut bid_file = BufReader::new(file);
        ciphers = bincode::deserialize_from(&mut bid_file)?;


        // Call max array
        // Set timer
```

```
        let start_time_max_array = Instant::now();

        let max_array= identify_max_bids(&ciphers, &ksk, &bsk, &encoder, &sk_in)?;

        let end_time_max_array = Instant::now();
        let elapsed_time_max_array =
end_time_max_array.duration_since(start_time_max_array);
        println!("Elapsed time Max Array: {} ms", elapsed_time_max_array.as_millis());

        // Calculate the Winning Bid Value, and who made that bid

        // Format the bid
        let max_bid_value = (max_array[0]/10.0).floor() as f64;
        let mut max_bid_array = max_array.clone();
        // For all values in max_bid_array
        for i in 0..max_bid_array.len() {
        // Subtract the bid value
        max_bid_array[i]= max_bid_array[i]-max_bid_value*10.0;
        }

        // Print the bid value
        println!("Winning Bid Value: {}", max_bid_value);
        // Print who made the bid
        println!("Winning Bidder ID: {:?}", max_bid_array);

        fn round_to(value: f64, places: i32) -> f64 {
        let factor = 10.0_f64.powi(places);
        (value * factor).round() / factor
        }

        let value = 1.23456789;
        let rounded_value = round_to(value, 1);
        assert!(approx_eq!(f64, rounded_value, 1.2, ulps = 2));

        let file_path = "bid.bin";
        if let Err(e) = fs::remove_file(file_path) {
        eprintln!("Failed to delete file {}: {}", file_path, e);
        }

        Ok(())
}
```

Appendix D: creat_bid.rs

```
use homomorphic_min_max::*;
use std::io::BufReader;
use std::io;
use std::fs;
use bincode;

pub fn public_bid(sk_in: &LWESecretKey, encoder: &Encoder) -> Result<(), Box<dyn
std::error::Error>> {

        println!("Enter your bid (0-2): ");
        let mut input_line = String::new();
        io::stdin()
        .read_line(&mut input_line)
        .expect("Failed to read line");
        let bid_input: f64 = input_line.trim().parse().expect("Input not an integer");

        println!("Enter your ID number (1-9): ");
        let mut input_line = String::new();
        io::stdin()
```

```
        .read_line(&mut input_line)
        .expect("Failed to read line");
        let id_input: f64 = input_line.trim().parse().expect("Input not an integer");

        //Combine the message nad the bid value.

        //The bids can be between 0-2, and the ID can be between 1-9
        let message: f64 = bid_input*10.0+id_input;

        // encrypt the messages
        let cipher = match LWE::encode_encrypt(sk_in, message, &encoder) {
        Ok(c) => c,
        Err(e) => {
                println!("An error occurred: {:?}", e);
                return Err(Box::new(e));
                }
        };

        let file_path = "bid.bin";

        // Check if file exists
        let data: Vec<LWE> = if fs::metadata(file_path).is_ok() {
        // Try to read and deserialize the file
        let file = fs::File::open(file_path).expect("Failed to open file");
        let mut buf_reader = BufReader::new(file);
        match bincode::deserialize_from(&mut buf_reader) {
                Ok(content) => content,
                Err(_) => Vec::new(), // If deserialization fails, initialize with an
empty Vec
        }
        } else {
        Vec::new() // If file doesn't exist, initialize with an empty Vec
        };

        // Add the new value to the array
        let new_value = cipher;
        let mut updated_data: Vec<LWE>= data;
        updated_data.push(new_value);

        let bid_file =
std::io::BufWriter::new(std::fs::File::create(file_path).unwrap());
        bincode::serialize_into(bid_file, &updated_data).unwrap();

        Ok(())
}
```

Appendix E: lib.rs

```
pub use concrete::*;

// START BORROWED CODE [29]
pub fn compute_max_min(cipher_1: &LWE, cipher_2: &LWE, ksk: &LWEKSK, bsk: &LWEBSK,
encoder: &Encoder)
        -> Result<(LWE, LWE), CryptoAPIError>
{

        // difference between the two ciphers
        let cipher_diff = cipher_2.sub_with_padding(&cipher_1)?;

        // programmable bootstrap to check if the difference is positive
        let mut cipher_diff_pos = cipher_diff.bootstrap_with_function(bsk,
                                                          |x| if x >= 0. { x } else
{ 0. },
```

```
                                                                encoder)?;

        // change the key back to the original one
        cipher_diff_pos = cipher_diff_pos.keyswitch(ksk)?;

        // add the result to cipher_1
        let mut result_max = cipher_1.add_with_padding(&cipher_diff_pos)?;

        // subtract the result from cipher_2
        let mut result_min = cipher_2.sub_with_padding(&cipher_diff_pos)?;

        // reset the encoder
        result_max = result_max.bootstrap_with_function(bsk, |x| x, encoder)?;
        result_min = result_min.bootstrap_with_function(bsk, |x| x, encoder)?;
        result_max = result_max.keyswitch(ksk)?;
        result_min = result_min.keyswitch(ksk)?;

        Ok((result_max, result_min))
}
// END BORROWED CODE [29]

pub fn sort_for_max(ciphers: &[LWE], ksk: &LWEKSK, bsk: &LWEBSK, encoder: &Encoder,
round: usize)
        -> Result<Vec<LWE>, Box<dyn std::error::Error>>
{

        // if ciphers contains less than two elements, just return it
        if ciphers.len() < 2 || round >= ciphers.len()-1 {
        return Ok(ciphers.to_vec())
        }

        let mut results = ciphers.to_vec();
        // Compare values from left to right, swapping always putting larger on right
        // Will not result in a fully sorted vector, but will end up with the larges
value at the end
        for i in 0..(ciphers.len()-1-round) {
        let (c_max, c_min) = compute_max_min(&results[i], &results[i+1], ksk, bsk,
encoder)?;
        results[i] = c_min;
        results[i+1] = c_max;
        }

        Ok(results)
}

pub fn identify_max_bids (ciphers: &[LWE], ksk: &LWEKSK, bsk: &LWEBSK, encoder:
&Encoder, sk_in: &LWESecretKey)
        -> Result<Vec<f64>, Box<dyn std::error::Error>>
{
        // RUN THIS UNTIL ALL DUPLICATE BIDS HAVE BEEN DETERMINED
        // Create max array
        let mut round: usize=0;
        let mut max_array: Vec<f64> = Vec::new();

        // Copy ciphers into results array
        let mut results = ciphers.to_vec();


        // To begin, call sort_for_max for the full array.
        // The largest value will end up in the right-most position
        results = sort_for_max(&results, ksk, bsk, encoder, round)?;

        // Decrypt that value and see what it is. Remove the voter ID
```

```
	let temp_master_max_value =
(&results[ciphers.len()-1].decrypt_decode_round(sk_in)?).round();
	// Add (with voter ID, to the max array)
	max_array.push(temp_master_max_value);
	let master_max_value=remove_voter_id(temp_master_max_value)?;

	// Encrypted master_max_value is the highest bid, with no voter ID
	let master_max_value_enc = LWE::encode_encrypt(sk_in, master_max_value,
&encoder)?;

	let mut done = false;
	round+=1;
	// Then, while not done:
	while !done && round < ciphers.len(){
	// Call sort_for_max for the array, but only for the left-most n-1 values
	results = sort_for_max(&results, ksk, bsk, encoder, round)?;

	// Call compute_max_min on the right-most minus n value and the re-encrypted
max without voter ID.
	let (temp_max, _temp_min) = compute_max_min(&results[ciphers.len()-1-round],
&master_max_value_enc, ksk, bsk, encoder)?;
	let temp_max_dec = temp_max.decrypt_decode_round(sk_in)?.round();
	if temp_max_dec > master_max_value {
		// If the max of this round is greater than the master_max_value, add
temp_max to the max_array. Continue the loop
		// Add temp_max to max_array
		max_array.push(temp_max_dec);
		round+=1;
	}
	else {
		// We're done
		done = true;
	}
	}

	Ok(max_array)

}

pub fn remove_voter_id(val: f64)
	-> Result<f64, Box<dyn std::error::Error>>
{
	// println!("Value: {}", val);
	let mut val = val.clone();
	val = (val/10.0).floor()*10.0;
	// print val
	// println!("Value: {}", val);
	Ok(val)
}
```