

HE Demo: Can we securely compute on encrypted data?

A dissertation submitted in partial fulfilment of
the requirements for the degree of
BACHELOR OF *SCIENCE* in Computer Science
in
The Queen's University of Belfast
by
Peter McGinley
28/04/2025

CSC3002 – COMPUTER SCIENCE PROJECT

Dissertation Cover Sheet

A signed and completed cover sheet must accompany the submission of the Software Engineering dissertation submitted for assessment.

Work submitted without a cover sheet will **NOT** be marked.

Student Name: Peter McGinley Student Number: 40327574

Project Title: HE Demo: Can we securely compute on encrypted data?

Supervisor: Dr Ciara Rafferty

Declaration of Academic Integrity

Before submitting your dissertation please check that the submission:

1. Has a full bibliography attached laid out according to the guidelines specified in the Student Project Handbook
2. Contains full acknowledgement of all secondary sources used (paper-based and electronic)
3. Does not exceed the specified page limit
4. Is clearly presented and proof-read
5. Is submitted on, or before, the specified or agreed due date. Late submissions will only be accepted in exceptional circumstances or where a deferment has been granted in advance.

By submitting your dissertation you declare that you have completed the tutorial on plagiarism at <http://www.qub.ac.uk/cite2write/introduction5.html> and are aware that it is an academic offence to plagiarise. You declare that the submission is your own original work. No part of it has been submitted for any other assignment and you have acknowledged all written and electronic sources used.

6. If selected as an exemplar, I agree to allow my dissertation to be used as a sample for future students. (Please delete this if you do not agree.)

Student's signature

Peter McGinley

28/04/2025

Acknowledgements

I would like to express a massive thanks to Dr Ciara Rafferty, for her invaluable support and guidance throughout this project. Ciara provided me with many different overviews and solutions that have helped me to develop it into the final system it has become.

Abstract

Fully Homomorphic Encryption (FHE) enables computation on encrypted data without ever needing to decrypt it. This project investigates whether FHE can enable a practical sealed-bid auction where bids remain encrypted at all times, ensuring complete confidentiality. An auction system was developed in Rust, using an FHE scheme to encrypt all bids. The system then homomorphically calculates both the highest bid (winning bid) and the average of the final bids placed by the bidders, meaning that the bids never have a need for being decrypted. Only the final results of the highest bid and average bid are decrypted. The design also implements tie-break rounds for when two or more bidders have an equal highest bid. This work showcases the feasibility of computing on FHE-encrypted data in a realistic real-world case.

Table of Contents

1. Introduction and Problem Area	5
1.1 Background on Homomorphic Encryption	5
1.2 Problem Statement	6
2. System Requirements and Specification	8
2.1 Functional Requirements	8
2.2 Non-Functional Requirements	9
3. Specification and Cryptographic Context	11
3.1 System Model	11
3.2 Data Representation	11
3.3 Cryptographic Operations	12
4. Design	13
4.1 System Architecture Overview	13
4.2 Homomorphic Computation of Maximum (Highest Bid)	13
4.3 Homomorphic Computation of Average	15
5. Implementation	17
5.1 Technology Stack and Libraries	17
5.2 Key Generation and Encryption	17
5.3 Input Handling	19
5.4 Computing the Highest Bid	19
5.5 Computing the Sum and Average	19
5.6 Decryption and Output	20
5.7 Testing and Debugging Aids	20
5.8 Code Structure	21
6. Testing	24
6.1 Unit Testing for Cryptographic Functions	24
6.2 Integration Testing of Auction Flow	24
6.3 Performance Testing	26
6.4 Security Testing	27
7. System Evaluation and Experimental Results	29
7.1 Meeting Functional Requirements	29
7.2 Analysis of Results of Implications	30
7.3 Future Improvements	32
7.4 Conclusion of Evaluation	32
8. References	33
9. Appendices	34
Appendix A: User Manual	34
Appendix B: Test Results and Output Logs	37
Appendix C: Code Repository and Technical Documentation	42

1. Introduction and Problem Area

In a sealed-bid auction, determining the winner and other outcomes must be done without revealing the losing bids to preserve bidder privacy [2]. Usually, this would require trust in a middle-man(auctioneer) or complex cryptographic protocols to keep bids unrevealed [3]. Fully Homomorphic Encryption (FHE) offers a potential solution by allowing encrypted data to be computed on. It permits operations on ciphertexts, so that the results returned are identical to the expected results from performing the same calculations on the original plaintext [4]. Essentially, this means an auctioneer (or computing service) could calculate the highest bid or average without seeing the plaintext of any individual bid.

Homomorphic encryption therefore overcomes the limitation of conventional encryption schemes which tend to rely on the decryption of data before processing [5]. The best way to describe capability of FHE would be to imagine a mixed-up Rubik's cube locked in a box, and without opening the box, you still are able to solve it [5]. With regards to auctions, the need for revealing phases or trust in the auctioneer could be eliminated [5].

1.1. Background:

The concept of homomorphic encryption has been found to first be theorised as early as 1978, but a fully homomorphic scheme wasn't first realised until 2009 by Craig Gentry [8]. Gentry's breakthrough proved it was possible to "compute arbitrary functions on encrypted data", paving the way for practical secure computing. FHE is now known to be the "holy grail" of cryptography as it provides a solution for what may be the final gap in data security by enabling data to remain encrypted during processing [5]. This demonstrates that sensitive information can be outsourced for computation (e.g. to a cloud service) without compromising data confidentiality.

Despite the promising concept of FHE, it carries significant challenges. Early FHE schemes were computationally impractical as operations were millions of times slower compared to plaintext computation [5]. Even today, performing non-trivial computations on encrypted data remains computationally expensive [7]. Due to the large computational overhead and memory usage, FHE is still a rarely implemented form of encryption [6]. Nevertheless, there have been major improvements in the efficiency of FHE within the last decade through the use of techniques such as bootstrapping (refreshing ciphertexts to reduce noise) being optimised so that modern schemes such as TFHE can refresh an entire encrypted bit in a fraction of a second [10]. There have been

many new libraries and tools, such as Google's FHE transpiler and Zama's Concrete library, that are making FHE more accessible for the use in practical experiments on encrypted data [5].

1.2. Problem Area:

Secure auctions have been an important area for cryptography. Prior research yielded in privacy-preserving sealed-bid auction, using Partial Homomorphic Encryption (PHE) combined with protocols like zero-knowledge proofs [3]. This resulted in an additive homomorphic scheme (such as Paillier encryption) which helped to compute the winning bid or totals through the use of addition without needing to decrypt individual bids, and interactive proofs ensured the auctioneer's correctness [3]. However, these approaches can be complex to implement and would more than likely still require trusted parties.

Exclusively, this dissertation investigates "Can we securely compute on fully homomorphically encrypted data?" through the implementation of FHE within an auction system. The main goal is to demonstrate a prototype that enables the computation of an auction's outcome while all bids remain encrypted. The system needs to calculate the maximum bid (to determine the auction winner) as the average bid (additional statistic) without the decryption of any bids, only allowing for the decryption and output of the final results of the calculations. Achieving this outcome would require solving several sub-problems: representing bids in a form suitable for FHE, performing comparison, addition and division operations on ciphertexts, and handling cases like tie bids, all while managing the limitations of current FHE technology.

This project focuses on a first price sealed-bid auction model: each bidder is allowed to submit one secret bid, which must be higher than the minimum bid set initially in the start-up of the system. This would mean bidders submit their best bid, allowing for a more competitive auction within a shorter period of time. All bidders' identities remain highly concealed throughout the process as well as winning bids remaining confidential. By using FHE, even the entity running the auction server would not be able to see any bid values during the computations.

In summary, this dissertation's problem domain intersects cryptography and auction theory. It contributes a case study on integrating advanced cryptographic techniques (fully homomorphic encryption) into an auction application. The design and implementation of FHE into this auction system through the use of Rust, allow for the analysis of FHE's correctness, security properties, and

performance. By doing so, we demonstrate the current capabilities and limitations when faced with the task of securely computing on fully homomorphically encrypted data in a realistic scenario.

2. System Requirements and Specification

This section defines the requirements and specification of the FHE-based auction system. The goal of this project is to create a secure auction application that preserves bid privacy while receiving the expected results from the calculations performed. The requirements are divided into functional requirements (what the system should do) and non-functional requirements (quality attributes and constraints), followed by a specification of the systems operating environment and cryptographic assumptions.

2.1. Functional Requirements:

1. **Auction Initialisation:** *The system should allow the auction administrator to set a minimum bid to ensure that there is a reserve price that all bids must meet or exceed.*
2. **Bid Submission:** *The system should accept bids from multiple users by requesting for users to enter their unique ID, which when entered correctly, prompts for a bid in the form of a whole number to be entered.*
3. **Input Termination:** *The bidding phase should continue until either all stored users have placed a bid or the termination character ('x') is entered as a unique ID.*
4. **Bid Encryption:** *Each bid should be immediately encrypted using the fully homomorphic encryption public key. The plaintext value will never be stored at any point of auction. Only ciphertext should be stored for processing.*
5. **Homomorphic Computation of Highest Bid:** *The system should compute the highest bid while all bids remain encrypted, determining the results of the highest bid through calculations performed on the ciphertext values. The algorithm should then output a ciphertext that matches the highest bid value.*
6. **Homomorphic Computation of Average Bid:** *The system should compute the average of all bids while they remain encrypted, determining the results of the highest bid through calculations performed on the ciphertext values. The algorithm should then output a ciphertext of the average bid.*
7. **Results Decryption:** *Only the final results, the highest bid and the average bid, should be decrypted using the FHE private key and output to the auction administrator or user. No other values should be decrypted during the process.*
8. **Tie Handling:** *If two or more bidders are found to have submitted identical highest bids (tie for first place), the system shall handle this by initiating a tie-break auction round, allowing all tied users to submit a new bid each that exceeds the previous bid that they placed. These tie-break rounds should continue until a highest bidder is found or the tie break runs 10 times*

in which case the auction will output the joint highest bid and will prompt for the system to be restarted and the auction ran with that highest bid as the new minimum bid and all users are able to bid again.

9. **Result Reporting:** *The system should report the auction results at the end of the process. At minimum it will report the highest bid value and average bid value. These are presented in plaintext, after decryption, to state the conclusion of an auction. No other bid values or information should be disclosed.*
10. **User Feedback and Validation:** *The system should validate inputs and provide feedback such as rejecting any bids below the minimum bid and prompting the user to enter a bid meeting or exceeding the specified minimum bid or if an incorrect unique user ID is entered, the system should reject the ID and prompt the user to enter a valid one.*

2.2. Non-Functional Requirements

1. **Security & Privacy:** *The primary security requirement should be bid confidentiality. No unauthorised party (including the auction server) should ever be able to view any individual bid. The homomorphic scheme should be semantically secure so that ciphertexts do not leak information. The private key should be protected, allowing only the component that performs the final decryption should hold the private key.*
2. **Correctness:** *The computation on the data should yield correct results. When the ciphertext of both the highest bid and average bid are decrypted, the values must equal the results from performing the same calculations on the original plaintext. The system should be rigorously tested to verify that homomorphic operations do not introduce errors.*
3. **Performance:** *The system should perform the auction computations within a reasonable timeframe for a prototype. In this case, using a test case of 10 bids, the system should not take any longer than a few minutes to complete all computations. To ensure efficiency, the main sections that should be focused on are the encryption parameters, limiting bit-length to reduce complexity and minimising the number of homomorphic operations. Memory usage is also to be considered as this system should be usable on any typical personal computer with a minimum dataset of 20 bids.*
4. **Usability:** *The system should be user-friendly for both the bidders and auctioneer. The bidder's perspective should be the ability to enter a unique ID and place a bid when prompted, as well as being able to place bids in tie-rounds when 2 or more highest bids are equal. The flow of the system should be easy to follow, with clear prompts and instructions. As for the auctioneer's perspective, they should have a straightforward system setup for*

completing tasks such as setting the minimum bid, starting the auction manually and seeing results. The system should also provide prompting of different sections of the auction process, such as when the bidding starts, the auction starts, and if there is any tie-rounds.

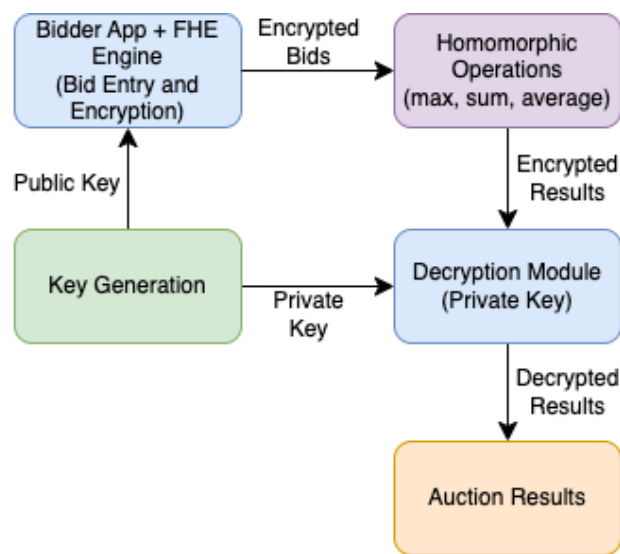
5. **Maintainability and Modularity:** *The software should be written in a clear and modular way. Cryptographic operations should be encapsulated (i.e. in separate functions), so that parameters can be tuned without affecting higher-level logic. Clear documentation must be presented in the form of a user manual so that future developers and/or researchers can understand the design and implementation of the system (the code).*
6. **Compatibility and Deployment:** *The system should be able to run on any platform that supports the language in which it is coded. It should also use a publicly available FHE library so that the project can be built and ran by others. A code repository should be provided, including the system code as well as any configurations for the FHE library.*
7. **Scalability (Future Consideration):** *While this prototype will only be expected to handle a limited number of bids, it is worth to note if the design could scale. Essentially, adding more bidders should not require any changes in the algorithm, but instead should linearly increase with the number of operations.*

3. Specification and Cryptographic Context

System Model:

The auction system is implemented as a command-line application Rust, due to its high performance and outstanding ecosystem (cargo). The prototype operates in a single-process, single-machine environment. The system should consist of the computation logic (encryption and homomorphic operations), an output stage (decryption and displaying of results) and an input interface (for bidders to enter bids). Below is a figure which illustrates the data flow and architecture of the system.

Figure 1:



The system uses a **public-key FHE scheme**. In this system's implementation, the scheme incorporates an existing FHE library (as detailed in the Implementation section). During the setup of the system, a keypair (public and private key) is generated. The private key is retained only to decrypt the final results. The system is treated as a threat model where the computation is honest-but-curious. It will correctly perform operations but is considered as untrusted regarding data privacy, therefore only being provided with encrypted data.

3.1.Data Representation:

The bids are being read as integers, with all bids and computed results lying within a representable range of chosen FHE parameters. For example, if using the FHE scheme over integers modulo some q , the bid values should be significantly smaller than q to avoid wrap-around or overflow. In this case, a reasonable upper bound on bids and encryption parameters was chosen accordingly. The average bid, if not already an integer, uses scaling to ensure output of an integer. In this

implementation, we output the floor of the average (as an integer) or an exact average if the sum of the bids divides evenly, avoiding dealing with fractional ciphertexts.

3.2. Cryptographic Operations:

The public library TFHE-rs is implemented as it supports three main homomorphic operations that the auction relies on:

- **Addition:**

Ciphertexts that encrypt 32-bit integers can be added directly with the overload '+' operator. The library refreshes noise automatically using programmable bootstrapping [5].

- **Comparison/Maximum Selection:**

TFHE supplies a constant-time *max* combinator for its unsigned-integer types. To get a winner, a linear scan is performed, repeatedly assigning:

$$(1) C_{\max} \leftarrow C_{\max} * \max(C_i)$$

which is equal to evaluating a "greater-than" circuit on encrypted operands but is exposed as a single API call. As for tie-detection, *enc_bid.eg(current_max)* and decrypt only that one-bit flag to identify bidders that match the encrypted maximum. This ensures that all bid values remain private apart from revealing minimum information.

- **Plaintext-Constant Division:**

TFBE-rs allows for division of an integer ciphertext by a clear-text *u32* with the operator '/'. Internally this multiplies by the Montgomery-pre-computed reciprocal and rounds *towards minus infinity*, resulting in:

$$(2) C_{\text{avg}} = []$$

Compared to an earlier design-note, we no longer need the implementation of modular inverse as the quotient is produced as a FHE ciphertext and is only revealed at the end of the auction, ensuring the privacy of individual bids.

4. Design

Designing an auction system that operates on fully homomorphically encrypted data required careful consideration of both software and cryptographic architecture aspects. This section goes into depth of the design of the system, specifically the system architecture, the homomorphic computation strategies for the average and highest bid and the structure of the tie-breaking mechanism. The main algorithms and dataflows are also highlighted through the use of diagrams and tables to illustrate how encrypted data is handled.

4.1 Architecture Overview

The system follows a **client-server model** in logical terms, where the *clients* are the bidders who encrypt their bids, and the *server* is the auction program that computes on the encrypted bids. In our implementation, the bidders and the server operations are encapsulated in one interactive application for simplicity, but architecturally we separate the roles of: (1) data encryption at the client side, (2) secure computation at the server side, and (3) decryption of results by an authorised party (the auctioneer). Figure 1 (shown earlier) captures this architecture. Each bidder uses the public key to encrypt their bid locally; the encrypted bids are then provided to the server which runs the FHE operations. Finally, the server (or the auctioneer's system) uses the secret key to decrypt the outcome.

To manage this as a software, the program is structured into modules:

1. An **Input module** handles user interaction: prompting for bids, reading input, validating input, and invoking the encryption function on each bid (main.rs).
2. A **Crypto module** contains the FHE library calls and key management. This module initialises the keys and provides functions such as addition and multiplication (Cargo.toml).
3. A **Computation and Output module** implements the logic to calculate the average bid and the highest bid. It utilises the crypto modules to operate on ciphertexts. This is also responsible for the decrypting and displaying of the final results (auction.rs).

This separation ensures the design is modular ensuring ease of maintenance, readability, usability and enhances scalability.

4.2 Homomorphic Computation of Maximum (Highest Bid)

Finding the maximum value among encrypted bids is one of the most challenging parts of this design as comparison is not a simple homomorphic operation. A ciphertext by itself carries no immediately

accessible information about the magnitude of its underlying plaintext [5]. In other words, given two encrypted bids, $C_a = \text{Enc}(a)$ and $C_b = \text{Enc}(b)$, one cannot simply tell which of a or b is larger without decrypting them. We must construct a circuit that computes something along the lines of $\text{Enc}(\max(a,b))$ using homomorphic operations such as addition and multiplication.

Approach:

We adopt a pairwise comparison strategy using a bit-wise comparison circuit. The main idea is to compare two encrypted integers, a and b , by evaluating $(a-b)$ in encrypted form and extracting the sign of the results. Through the use of an encrypted bit that is 1 if $a > b$ and 0 otherwise, then a selector could be used to pick a or b . Many designs for homomorphic max use a formula like:

$$(3) \max(a,b) = a * \text{isGreater}(a,b) + b * (1 - \text{isGreater}(a,b))$$

where $\text{isGreater}(a,b)$ is an indicator (1 if $a > b$) [5]. The implementation of $\text{isGreater}(a,b)$ can be done by computing $(a-b)$ and then checking its sign. Checking the sign can be reduced to analysing the most significant bit of $(a-b)$ in two's complement representation.

Our design encodes integers in binary form. Supposing each bid is represented with L bits, allowing for ciphertexts for each bit of each bid. To compare a and b , we perform a homomorphic subtraction bit by bit to get the difference, $d = a - b$, in two's complement. The result, d , has a MSB (Most Significant Bit) which acts as a sign bit. If d is negative (indicating $a < b$) then the MSB will be 1 and if d is non-negative (indicating $a \geq b$) the MSB will be 0, then using this value to perform $(1 - \text{MSB})$ as $\text{isGreater}(a,b)$. We use greater than or equal to for the non-negative is due to an exact equality only being needed for tie-handling which is done separately. In summary, the comparator circuit produces an encrypted bit, $g = \text{Enc}(1)$ if a is greater than or equal to b , or $g = \text{Enc}(0)$ if b is greater than a , meaning the max in encrypted form is:

$$(4) C_{\max} = g * C_a + (1-g) * C_b$$

All operations used to form this encrypted formula (subtraction, bit operations, multiplication) are done within the FHE scheme, enabling C_{\max} to be a result of a fully homomorphic operation of identifying the larger plaintext value as a ciphertext.

We designed this to minimise depth and number of bootstrapping operations. Each comparison of two L -bit numbers uses $O(L)$ homomorphic binary gates. By using bootstrapping enabled FHE (like TFHE), each gate can refresh the ciphertext, meaning the circuit depth is not a limiting factor except in terms of time. TFHE-rs by Zama, which is implemented in this system, provides programmable bootstrapping that can even implement a comparison directly [5]. Prior to work using Concrete has demonstrated a sorting algorithm on encrypted data by iteratively applying a compare and swap on the ciphertext pairs [5]. This demonstration was used as inspiration for system, implementing a simplified version more suitable to this system.

Max of N values: To get the maximum of more than two bids, a tournament or linear scan of comparisons can be performed. Using a linear approach starts with an encrypted variable, C_{\max} , initialised to the first bid's ciphertext, updating C_{\max} with every comparison:

$$(5) \ C_{\max} = \text{EncMax}(C_{\max}, C_i)$$

After the processing of all bids, C_{\max} will contain the encryption of the highest bid among the set. This approach requires $N-1$ pairwise comparisons for N bids [5]. The linear method was chosen as it is straightforward.

Tie consideration:

The above comparator returns a if a is greater than or equal to b , in which case, $a=b$ effectively returns a . This means that if two or more bids are exactly equal, whichever bid is discovered in the in the linear scan last will end up as the highest bid, nullifying any equal bids. Normally in auctions if two bids are equal, it is a tie and that would be resolved by another round instead of arbitrarily picking one of the bids. The solution for this issue is discussed in more detail further into the design section.

4.3 Homomorphic Computation of Average

Computing the average of a set of encrypted bids presents a distinct challenge, due to the lack of native support for division within most FHE schemes. Although addition is straightforward and widely supported across most FHE schemes, division usually requires workaround techniques. In this case, the average is defined as:

$$(6) \text{ avg} = \frac{1}{N} \sum_{i=1}^N \text{bid}_i$$

where N is the total number of valid bids. The main challenge lies in securely computing this formula without decrypting any intermediate values.

Approach:

We begin computing the sum of all encrypted bids using homomorphic addition. We do this by casting each 16-bit encrypted bid (*FheUint16*) into a 32-bit encrypted integer (*FheUint32*) and summing them together. This would yield in a single ciphertext *sum_ct*, which encrypts the total of all bids. All summations occur under encryption without any data being revealed.

Division Method:

To calculate the average, the encrypted sum must be divided by the number of bids, N . In prior approaches using different schemes (like BGV or BGV), this often involved computing the modular inverse of N , provided $\gcd(N, M) = 1$, where M is the plaintext modulus [5]. However, implementing TFHE-rs enables the use of ciphertext-by-plaintext division.

This division is achieved through the operator $/$, where *sum_ct* / N performs encrypted floor division using N as a plaintext constant. This is possible due to TFHE-rs supporting homomorphic division by known scalars directly using programmable bootstrapping. Then result to be casted back to a 16-bit integer (*FheUint16*) for decryption and output.

In summary, the final method ensures full homomorphic encryption throughout the computation of the average bid, allowing only for decryption of the result when being output. TFHE-rs supports the homomorphic operations involving division by cleartext constants, which allows for the avoidance of complex modular arithmetic and enables clean and efficient implementation.

5. Implementation

The system has been implemented in Rust programming language, which was chosen for its strong safety guarantees and performance. Rust's package ecosystem (Cargo) and the availability of cryptographic crates make it convenient to integrate a fully homomorphic encryption library into the project. The core implementation resides in a source file, `main.rs`, which contains the program's entry point and orchestrates the auction logic described in the design.

5.1. Technology Stack and Libraries

Rust and Cargo: Rust (edition 2021) was used as well as Cargo being employed to manage dependencies and build the project. The code was written to be portable across operating systems, meaning no platform-specific APIs were used from the console I/O for interacting with users.

FHE Library: TFHE-rs by Zama was utilised, which is a Rust library implementing fully homomorphic encryption (TFHE-based). Concrete provides an API for generating keys and operating on ciphertexts.

This library was selected due to the fact that:

- It is a Rust-native library
- It supports both Boolean and small-integer homomorphic operations
- It has built-in functions or examples for common operations like addition and a way to implement comparison

In the cargo manifest (`Cargo.toml`), we include the dependencies:

```
[dependencies]
tfhe = { git = "https://github.com/zama-ai/tfhe-rs", branch = "main",
features = ["boolean", "shortint", "integer"] }
rand = "0.8"
crossterm = "0.25"
```

- TFHE: used to import the TFHE-rs library for homomorphic operations.
- Rand: Used to generate random bids for testing.
- Crossterm: Used to clear the console on all OS (without this, the clear console command only works on the specified OS).

The library requires setting certain feature flags for parameters (similar to using a particular security level and polynomial modulus degree for the underlying LWE scheme). We used default 128-bit

security parameter settings which come configured to handle a few dozen Boolean operations with bootstrapping.

5.2. Key Generation and Encryption

When the program starts up, the FHE key is generated. With Concrete, this typically involves choosing a secret key distribution and outputting a public key. The exact details are abstracted by the library through the use of a function within our code.

```
let (client_key, server_key) = gen_keys(config)
```

The library also chooses parameters such as polynomial degrees, noise budgets, etc. Since this application is being ran locally, the system contains both the public aspect (used for homomorphic operations) and the secret aspect (for decryption) due to the system running locally.

The encryption of the bid is done when it gets added to the system:

```
pub fn addBid(&mut self, user_id: &str, bid_value: u16) {  
  
    // Encrypt the bid using the ClientKey. This produces a FHE  
    ciphertext.  
    let enc_bid = FheUint16::encrypt(bid_value, self.client_key);  
    // Insert or update the bid in the HashMap  
    self.bids.insert(user_id.to_string(), enc_bid);  
}
```

TFHE-rs supports different integer widths as well as Boolean encryption for individual bits. The integer range 16-bit (short-int) was chose due to this system only working with small bids. This function simply takes the input from the user, encrypts it and maps it to the user from an array.

5.3. Input Handling

The main logic (within `main.rs`) typically uses standard Rust I/O to collect bids. Firstly, a minimum bid (`min_bid`) in plaintext (only used for validation). Then we repeatedly prompt:

Enter bidder ID (or 'x' to finish):

If 'x' is entered, the bid collection ends, otherwise we parse the user ID and then ask:

Enter bid for [ID]:

We then parse the user's numeric input (`u16`), to verify that it is at least the `min_bid`, and call:

```
auction.add_Bid(userid, bid_value);
```

Inside `add_bid`, the `bid_value` is encrypted with the client key and stored resulting ciphertext (`FheUint16`) in a `HashMap<String, FheUint16>` keyed by `user_id`. No single plaintext value remains in memory after encryption. This loop will continue until either all bidders are input or when 'x' is entered as a user ID.

5.4. Computing the Highest Bid

The highest bid is computed within `auction.rs`, within the function:

```
pub fn compute_max_encrypted_and_top(&self) -> (FheUint16, Vec<String>)
```

The maximum is computed in 2 steps:

1. Iterate over all encrypted bids using, `current_max = current_max.max(enc_bid);`
2. Compare each bid to `current_max` to see what user(s) match that max

For comparison, TFHE-rs provides:

```
let is_equal = enc_bid.eq(&current_max);
if is_equal.decrypt(self.client_key) {
    top_users.push(user.clone());
}
```

- `enc_bid.eq(¤t_max)` creates a homomorphic Boolean ciphertext.

Decrypting this Boolean reveals either *true* or *false* without exposing the numeric bid, allowing the system to identify the bid which matched the maximum. This means that if multiple user's bids

match the max, there will be a tie scenario. We can prompt only those who match the highest bid to rebid, recomputing the max bid with those new values, and eventually yielding a single winner.

5.5. Computing the Sum and Average

When computing the average, the code uses 32-bit encrypted integers to avoid overflow while summing multiple 16-bit bids and converts the count into a clear-text constant:

```
let count = self.bids.len() as u32;

let mut sum_enc = FheUint32::encrypt(0u32, self.client_key);
for enc_bid in self.bids.values() {
    let enc_bid_32 = FheUint32::cast_from(enc_bid.clone());
    sum_enc = sum_enc + enc_bid_32; // homomorphic addition
}
```

After homomorphically summing in `sum_enc`, the result remains encrypted. The code then moves onto dividing the encrypted sum by the count:

```
let mut sum_enc = FheUint32::encrypt(0u32, self.client_key);
for enc_bid in self.bids.values() {
    let enc_bid_32 = FheUint32::cast_from(enc_bid.clone());
    sum_enc = sum_enc + enc_bid_32;
```

This computation is then used to output a decrypted plaintext value of the average of all bids. Also all fractional averages are rounded down to ensure the output of an integer.

5.6. Decryption and Output

Finally, after computing results, they are decrypted to obtain plaintext for display. For example, to show the highest bid:

```
let (max_ct, top_users) = auction.compute_max_encrypted_and_top();
let max_value: u16 = max_ct.decrypt(auction.client_key());
println!("Highest bid = {}", max_value);
```

In summary, the TFHE-rs library supports each step, from encrypting a 16-bit bid to homomorphically determining the maximum and summing for the average. The final results are decrypted and displayed, ensuring no individual plaintext bid is exposed at any time.

5.7. Testing and Debugging Aids in Implementation

During development, several logging and debugging outputs have been recorded to ensure that the system is performing correctly. We tested the compare function on known outputs (.i.e. giving the

system bids of 100 and 200) and using the compare function to determine the expected output (i.e. 200). These unit tests resided in a shell script, where predetermined bids can be hard-coded to ensure the algorithm works as expected. There were also performance timings recorded such as the encryption of each bid, the time taken for the max computation to complete and so on. We found that the comparison circuits were the slowest section, unlike the summation which was relatively fast. This is due to the computational expense of the homomorphic operations required to calculate the highest bid.

Challenges Encountered

1. **Parameter Tuning:** The FHE parameters had to be balanced for correctness versus speed. If the parameters were set too small, the noise budget might be exceeded, which would cause incorrect results or decryption failures. However, if they're too large, performance would slow down. The bootstrapping approach implemented ensures correctness but is more computationally expensive. This trade-off was acceptable as the system could still perform on a standard laptop while providing correct results and allowed the system to not experience noise-related decryption errors as bootstrapping resets noise at each step [9].
2. **Data Type Limitations:** We treated the bids as 16-bit integers, meaning if someone entered a bid larger than the scheme is able to represent, it wraps around modulo 2^{16} . We documented that bids must lie within a specific range (e.g. 0 – 65535 for 16-bit). Alternatively, 32-bit integers could be used but they would come at the cost of additional bootstrapping overhead. Therefore, for this project, 16-bit integers were substantial to provide a sufficient demo.

Continuing the Implementation Details

Throughout coding, we adhered to Rust's ownership and safety rules. The ciphertext data structures are large, each ciphertext might include polynomial coefficients internally. To avoid unnecessary cloning, instead of creating new ciphertexts in the homomorphic max function, the current max gets updated. Rust's borrow checker helped ensure safe concurrent access, though in this prototype, the code was kept single-threaded for simplicity.

5.8. Code Structure

In the final implementation is subdivided into two main components, `main.rs` and `auction.rs`, together fulfilling the design principles previously outlined. Below is an overview of how both components align with the described structure:

1. Struct Definitions

In `auction.rs`, the primary data structure is:

```
pub struct Auction<'a> {  
    client_key: &'a ClientKey,  
    min_bid: u16,  
    bids: HashMap<String, FheUint16>,  
}
```

This structure references the secret key which is vital for encrypting and decrypting data while holding both the minimum bid (in plaintext) and the encrypted bids. The code specifically controls *FheUint16* objects from THFE-rs.

2. Key Initialisation

In `main.rs`, the program begins by generating the cryptographic keys:

```
let config = ConfigBuilder::default().build();  
let (client_key, server_key) = generate_keys(config);  
set_server_key(server_key);
```

This makes sure of the availability of a client key for encryption and decryption, and a server key enabling fully homomorphic operations. Using these keys, the system constructs an Auction object, passing the client key (along with a minimum bid) into the constructor.

3. Crypto Operation Functions

The `auction.rs` file contains cryptographic methods in the *Auction* implementation:

- **add_bid**: Encrypts each user's plaintext bid as *FheUint16* and stores it in the *bids* hash map.
- **compute_max_encrypted_bid**: Iterates over all encrypted bids and uses TFHE-rs's *max()* method to determine the highest bid homomorphically, returning ciphertext of the maximum and a list of users whose bids match the maximum (for tie detection)
- **compute_average_encrypted**: homomorphically sums all bids in a 32-bit ciphertext to prevent overflow (via *FheUint32::cast_from*), then divides the sum by the clear constant count of bids before casting the result the encrypted average back to a 16-bit ciphertext (via *FheUint16::cast_from*).

4. Main Flow

The core logic is located in *main.rs*, and orchestrates the entire auction process:

1. **Prompting for Input:** The user is prompted to supply a minimum bid, followed by a loop that gathers each valid user's ID and a corresponding bid. All bids are validated to ensure they exceed the minimum and then encrypted and stored via *auction.add_bid(...)*.
2. **Bidding Completion:** When bidding is completed, either by all valid user's having placed a bid or 'x' being entered as a user ID, the system calls *compute_max_encrypted_bid()* to find the maximum bid ciphertext and identify any ties.
3. **Tie-Break Rounds:** If the returned user list has more than one entry, tie-break logic is initiated. All users who had equal highest bids are prompted again to place a higher bid than their previous bid or withdraw. This process will repeat until there is only one highest bidder.
4. **Final Output:** The program decrypts the highest bid for display as well as calling *compute_average_encrypted()* to calculate the average and then decrypt the result. The results are then printed out by the program, concluding the auction.

The program has been thoroughly populated with inline comments to ensure ease of maintainability as well as clear guidance for anyone reading or modifying the code. This approach shows that Rust's safety features and TFHE-rs's fully homomorphic encryption can effectively work together. This code structure ensures that bid confidentiality is maintained at all times and only results required to be output are decrypted.

6. Testing

Note for unit and integration tests, small datasets and bid values are used to ensure fast computation for efficient testing

6.1. Unit Tests for Cryptographic Functions: A number of unit tests were written in Rust to validate the homomorphic operations in an isolated manner. For example, one of the tests, *compare_and_select*, encrypts known bids and identifies the highest value:

- Encrypt 200 and 100, homomorphically select maximum, decrypt result – expected result of 200.
- Encrypt 300 and 300, homomorphically select maximum, decrypt result – expected result of 300 and check for detection of tie.
- Encrypt a series of values, [100, 400, 250, 150], homomorphically select maximum, decrypt result – expected result of 400

For the calculation of the average, there were similar unit tests implemented:

- Encrypt a series of values [100, 250, 200, 350], homomorphically calculate average, decrypt result – expected result of 238 (rounded up)
- Edge case: Encrypt a single bid of 200, homomorphically calculate average, decrypt result – expected result of 100 to confirm successfully single-element handling.

All unit tests passed, confirming that the encryption and computations implemented were functioning correctly and that the decryption results are equal to the results expected of the same calculations done on the plaintext.

6.2. Integration Testing of Auction Flow:

Full program tests were also manually implemented to simulate different scenarios and ensure end-to-end correctness:

1. Basic No-Tie Scenario:

- **Scenario:** Two bidders (IDs User1 and User2) with distinct bids, User1 bidding 200 and User2 bidding 300.
- **Expected outcome:** Output results indicate 300 is the highest bid and the average bid is 250.

- **Final Results:** The program output the expected results verifying the integration of input, computation and output.

2. Tie Scenario:

- **Scenario:** Three bidders (IDs User1, User2 and User3) with distinct bids, User1 bidding 100, User2 bidding 250 and User3 bidding 250.
- **Expected Prompts:** Program initiates a tie round for User2 and User3 where User2 bids 300 and User3 bids 325.
- **Expected Outcome:** Output results confirm 325 as the highest bid and 242 as the average.
- **Final Results:** The program initiated a tie rounds for the correct users and was able to compute and output the correct highest bid and average.

3. All Bidders Tie:

- **Scenario:** Four bidders (IDs User1, User2, User3 and User4) with distinct bids, each user bidding 300.
- **Expected Prompts:** Program initiates a tie round for all users where User1 and User2 enter 350, while User3 and User4 enter 400. A second tie round is initiated for User3 (rebids 450) and User4 (rebids 425).
- **Expected Outcome:** The highest bid is output as 450 and the average is output as 394.
- **Final Results:** The program initiated both tie rounds for the correct users and was able to compute and output the correct highest bid and average.

4. Validation of Minimum Bid

- **Scenario:** An invalid bid is input (below the minimum) for one users (User1) and then leaves the bid empty – minimum bid set at 200, both users bid 150.
- **Expected Prompts:** Program outputs prompts indicating that the bid entered is below the minimum bid and asks for a User ID again. When User1 goes to place another bid and leaves it empty, the program prompts indicating it is an invalid bid.
- **Final Results:** The program displays the correct prompts and doesn't allow any empty bids or bids under the minimum.

5. No Bids

- **Scenario:** No bids are placed and 'x' is entered to start the auction.
- **Expected Prompts:** Program outputs prompt stating that no bids were placed and the auction was terminated
- **Final Results:** The program displays the correct prompts and terminates the auction.

6.3. Performance Testing

Performance testing was carried out to evaluate the efficiency of the auction system, which was assessed through automated benchmarking as opposed to using load-testing tools. The main outcome was to ensure the computational complexity and runtimes matched the design expectations. The key outcomes were as follows:

1. Key Generation:

- Consistently remained around 1 second with no correlation to the number of bidders, confirming the expected fixed overhead for cryptographic setup.

2. Encryption Efficiency:

- The time taken for this process was minimal as it takes less than a millisecond, which verifies that the encryption was computationally inexpensive.

3. Homomorphic Computation:

- These homomorphic operations were the primary performance cost. The tests clearly outline a linear increase in runtimes, proportionate to the number of bidders. This confirmed the theoretical performance discussed during the design phase, particularly the impact of sequential homomorphic operations such as comparisons.

4. Decryption Efficiency:

- The time taken for this process was virtually instantaneous, consistently recording negligible runtime across all tests.

During the test, all bids assigned a random value between 1000 and 50000. Although the system utilises smaller bids within the hundreds range, testing with these higher values ensures that the system would be capable of handling larger datasets. The test is also done on different sets of bidders, 10, 20 and 50.

Process	10 Bidders	20 Bidders	50 Bidders
Key Generation (ms)	1075	1042	1006
Encrypt + Store (ms)	8	17	40
Max + Average Comp (ms)	11359	17855	44602
Decrypt (ms)	0	0	0
Highest Bid (final value)	47243	49723	49777
Average Bid (final value)	23466	26335	30122

As can be seen in the table above, the time taken for encryption and homomorphic computations increases linearly with number of bids. The decryption process is so fast due to only the highest bid and average bid being decrypted, therefore since it takes less than a millisecond, there is no accurate measurement for how long the decryption process takes.

Overall, the performance testing ensured that the system meets the performance expectations, mainly for medium sized auctions. The predictions made with regards to the computational cost associated with homomorphic computations was predictable and matched theoretical models, confirming the design choices were appropriate.

6.4. Security Testing

Security of the system was evaluated mostly through code review and reasoning, instead of an automated penetration testing method (as primary threat is improper decryption or leakage). Instead the information gathered from the security testing is as follows:

1. There is no accidental printing or storage of plaintext bids after encryption
2. All comparisons are performed in ciphertext and at no point does the program branch on plaintext values (timings could leak information). Every path in the homomorphic operations takes the same number of steps regardless of data, making sure the timing side-channel is minimised.

Finally, failure conditions were also tested. The main test was the decryption failing due to a value that exceeds the set parameters. There were no failures in normal operation due to the FHE parameters, however when parameters were intentionally reduced, a failure was forced. The decrypted result was not the expected result due to the small polynomial degree. This highlighted the importance of proper parameter selection to ensure correct results.

This testing highlights that the correctness and performance of the system is successful, and the outcome is what was expected during the design phase.

7. System Evaluation and Experimental Results

In this section, we evaluate the auction system against the requirements which were set in the beginning and discuss how the experimental results reflect on the feasibility of computing on fully homomorphic encrypted data. Taken into consideration are the security achievements, functionality, performance/scalability and usability. We also compare the outcome to expectations and identify the limitations and areas of future improvement.

7.1. Meeting System Requirements

Privacy and Safety: The system successfully kept all individual bids confidential throughout the computation. At no point during are the plaintext bids revealed to the ‘server’ or any entity other than the bidder who submitted it. The only outputs are the average bid and the average bid, which were intended to be displayed. This correlates with our main security goal; *no unauthorised party (including the auction ‘server’) should ever be able to view any individual bid.*

The system ensures integrity of the result as well. Due to all the computations being deterministic and based on inputs, any malicious attempt to tamper with the ciphertexts would result in incorrect final decryption that are easily detected (this would require breaking the encryption). The system also assumes that the auction server executes the protocol correctly (honest-but-curious model). At a larger scale, it would be wise to implement verifiability proofs to enhance trust. Overall the system demonstrates a high level of privacy and safety.

Functional Correctness:

All functional requirements were met as demonstrated by the test:

- The system correctly enforces a minimum bid.
- It collects bids until the auction is started and handles input errors correctly
- The highest bid and average bid results are correctly provided by the homomorphic computations.
- Tie-break rounds are identified and executed flawlessly, producing a single winner when ties are resolved.
- The required information is output (highest bid, average bid, tie-break information) and nothing more.

Performance:

The requirement of reasonable performance is met, but with limitations. Using a small number of bidders (in the range of 10-15 bidders), the system's runtime is on the order of seconds, which is acceptable in this context, however the performance degrades linearly with the more bidders involved, so scaling up to hundreds of bidders would result in impractically long computation times.

That said, the system could be optimised using parallel processing or a more efficient FHE scheme for comparisons could potentially speed it up considerably. Hardware improvements or deploying GPUs/ASICs for homomorphic operations could reduce runtime. For practical deployments in large auctions, the current standard of FHE technology would require additional optimisation [7].

Usability:

From a user perspective, the system is simple. In trial runs, bidders simply follow a series of text prompts and are provided with a formatted output. Instructions are also provided (Appendix A) that are sufficient to run the program. Due to everything occurring behind the scenes, bidders do not need to know that the encryption is happening as they just input their bids normally. The instructions also contain an example run through, so the bidder knows what to accept. Overall, the system remains user-friendly for a command-line application. The users are oblivious of anything going on behind the scenes (i.e. the homomorphic computations), therefore they are unaware of the homomorphic aspect of the system unless they are informed.

Robustness:

The system handles unexpected inputs and edge cases during testing, which indicates good robustness. For example, trying an extremely large bid (beyond configured range) resulted in an output, which although it was incorrect, provides proof that the system is robust due to it not crashing (this was instead documented as a limitation). The system was also tested to ensure the program doesn't crash on empty auctions or repeat tie-breaks.

7.2. Analysis of Results of Implications

The experiments confirm that Fully Homomorphic Encryption (FHE) can be used to compute both the average and maximum bid *without revealing any individual bid*. The key observations include:

- **Correctness under encryption:**

Homomorphic operations for max and average bids produced identical results to plaintext evaluation which validates the theoretical claim that “any computation can be performed on encrypted data” when circuits and parameters are chosen appropriately [2].

- **Realistic auction workflow:**

By combining arithmetic with non-linear comparison, it was demonstrated that a moderately complex workflow can remain encrypted end-to-end. This suggests feasibility for sealed auctions, privacy-preserving voting or statistical analysis.

Performance and Scalability – With up to a few dozen bids, the runtime of the system is relatively low, but due to computation growing linearly with the number ‘ N ’ of bidders, extrapolation indicates ≈ 1 minute for $N > 10^2$ and ≈ 10 minutes for $N > 10^3$. These latencies could cause runtime issues in a large-scale implementation. However, the broader literature notes that FHE remains computationally expensive but is improving steadily [7][6].

Ciphertext size & memory – An encrypted 16-bit bid occupies hundreds of kilobytes. Memory is modest for small N but grows quickly. SIMD-style ‘packing’ could store many bids per ciphertext [10]. TFHE’s bit-oriented design lacks this optimisation. The information below better explains how this actually works:

- One FheUint16 = 16 bits x 32KB/bit = 512KB per encrypted bid
- 10 bids \rightarrow ~5MB of RAM
- 20 bids \rightarrow ~50MB of RAM
- 50 bids \rightarrow ~500MB of RAM

For small demos (10 bidders), the required RAM is perfect, however if this was pushed to handle hundreds or thousands of bids at once, the system would require the implementation of a packing layer to allow multiple integers to be packed into a single ciphertext, reducing the required RAM.

Tie-break information leakage – Our tie-break loop discloses which bidders tied. A fully private design could trigger private client prompts or adopt protocols that avoid revealing ties but would complicate interaction. In many real auctions announcing a tie is acceptable.

Comparison with traditional protocols – Classic secure-auction designs use threshold decryption with partially homomorphic encryption plus trustee interaction [3]. By implementing FHE, the need

for trustees is removed but it causes a heavier computational cost. If computational power is cheaper the multi-party coordination, then FHE would be seen as the more attractive option.

7.3. Future Improvements

- **Parallel/Hardware acceleration** – Multi-threaded comparisons or GPU bootstrapping.
- **Wider plaintexts** – Extending to 32/64-bit bids and evaluating the overhead.
- **Packing Layer** – Implementing a packing layer would allow for multiple integers to be packed into a single ciphertext, reducing the memory required for the system to efficiently run.
- **Second price (Vickrey auctions)** – Homomorphically zero the max bid then recomputes a second max [9].
- **Alternate schemes** – Using CKKS for fixed-point averages or levelled FHE without bootstrapping for fixed circuits [5].
- **Verifiability** – Publish ciphertexts and allow any party to recompute the encrypted max, or attachment of a zero-knowledge proof that decryption is honest.

7.4. Conclusion of Evaluation

We set out to ask: “Can we securely compute on fully homomorphically encrypted data?” The answer is **Yes – with restrictions**. If the system was to be implemented as a functioning system for large auctions, a packing layer would need to be implemented to reduce the amount of required RAM for the system to run efficiently with a relevant dataset (hundreds/thousands of bids). Although this system is significantly slower than computing with partial encryption, it allows for computations the auction requires, to be ran while ensuring full confidentiality of all bids. The prototype auction process remained encrypted throughout the computations and yielded correct results, demonstrating the practical power of FHE. As stated, current limitations are mainly performance-related, aligning with the community’s view that FHE is ‘promising but still maturing’ [6]. As libraries and hardware advance, encrypted auctions of much larger scale will become increasingly feasible [5].

8. References

- [1] Github Code Repository - <https://github.com/PMCG03/ZamaHEBid/>
- [2] A. Moll, "Developing Blind-Bidding Auctions to Explore Fully Homomorphic Encryption," M.S. thesis, Dept. of Computer Sci., St. Cloud State Univ., MN, USA, 2023.
- [3] D. C. Parkes, M. O. Rabin, S. M. Shieber, and C. A. Thorpe, "Practical secrecy-preserving, verifiably correct and trustworthy auctions," in *Proc. 8th Int. Conf. Electronic Commerce (ICEC'06)*, Fredericton, Canada, Aug. 2006, pp. 70–81.
- [4] C. Danjou, "Onchain Blind Auctions Using Homomorphic Encryption and the fhEVM," *Zama Blog*, Jul. 10, 2023. [Online]. Available: <https://zama.ai/post/on-chain-blind-auctions-using-homomorphic-encryption>. [Accessed Apr. 2025].
- [5] R. Solomon, F. Michel, J. Wilson, and E. Cottle, "Max, min, and sort functions using programmable bootstrapping in Concrete FHE," *Optalysys Blog on Medium*, Mar. 24, 2022. [Online]. Available: <https://medium.com/optalysys/max-min-and-sort-functions-using-programmable-bootstrapping-in-concrete-fhe-ac4d9378f17d>. [Accessed Apr. 2025].
- [6] "Homomorphic Encryption: What Businesses Need to Know," *Hitachi Cyber Blog*, Apr. 4, 2025. [Online]. Available: <https://hitachicyber.com/homomorphic-encryption-what-businesses-need-to-know>. [Accessed Apr. 2025].
- [7] A. Zewe, "Security scheme could protect sensitive data during cloud computation," *MIT News*, Mar. 19, 2025. [Online]. Available: <https://news.mit.edu/2025/security-scheme-protect-data-cloud-computation>. [Accessed Apr. 2025].
- [8] "Homomorphic Encryption," *Chainlink Education Hub*, May 8, 2024. [Online]. Available: <https://chain.link/education-hub/homomorphic-encryption>. [Accessed Apr. 2025].
- [9] R. Solomon, "Building private verifiable auctions with FHE," *Sunscreen Tech Blog*, Nov. 8, 2023. [Online]. Available: <https://blog.sunscreen.tech/building-private-verifiable-auctions>. [Accessed Apr. 2025].
- [10] Wikipedia contributors, "Homomorphic encryption," *Wikipedia, The Free Encyclopedia*. [Online]. Available: https://en.wikipedia.org/wiki/Homomorphic_encryption. [Accessed Apr. 2025].

9. Appendices

9.1. Appendix A: User Manual

System Overview:

The FHE auction demo program allows an auction to be conducted so that all bids are kept encrypted throughout the auction. The user running the program inputs a minimum bid and collects from participants one by one. The highest bid and average bid are output at the end. Bidders only interact with the interface to enter bids.

Prerequisites:

- Rust 1.65+ installed (for compiling the code).
- The FHE library (TFHE by Zama) included via Cargo – this is handled by the provided code's Cargo.toml.
- A system with a reasonable CPU (multi-core recommended for speed, though the program is single threaded by default).

Setup and Compilation:

1. Unzip the provided repository ZamaHEBid.zip to a directory.
2. Navigate to the project directory in a terminal.
3. Run `cargo build --release`. This will download dependencies and compile the program in release mode. The output binary will be in `target/release/`. (For testing or development, you can use `cargo run` to compile and run in one step in debug mode).
4. Ensure you have permission to execute the binary if on UNIX (Cargo typically handles this).

Running the Program:

- Launch the program from a terminal: `cargo run --release`
- The program will prompt you step by step. Below is an example session (user input is shown after prompts):

```
*** Welcome to the Encrypted Auction CLI ***

Enter the minimum bid (whole number): 100

(console cleared)

Minimum bid set.

Enter 'x' at the User ID prompt to finish bidding early.

Please enter your user ID: User1

User1 - enter your bid (must be a whole number above 100):120
```

```

(console cleared)

Please enter your user ID: User2

User2 - enter your bid (must be a whole number above 100):140

(console cleared)

Please enter your user ID: User3

User3 - enter your bid (must be a whole number above 100):140

(console cleared)

Please enter your user ID: x

Bidding terminated early.

```

At this point, the program has collected 3 bids, [120,140,140]. It will then homomorphically compute the results which may take a few minutes to process. It then prints:

```

*** Tie detected! ***

Users [User2, User3] all bid 140.

Tiebreaker: these users must rebid to determine a single winner.

```

This indicates that both User2 and User3 have tied for the highest bid of 140. The program has now initiated a tiebreaker to collect new bids from only those who had the equal highest bids:

```

User2 - enter a new bid higher than 140 (or 'x' to withdraw):150

(console cleared)

User2 - enter a new bid higher than 140 (or 'x' to withdraw):160

(console clears)

```

User1 bids 150 while User2 bids 160 in the tiebreak. The program computes again and outputs:

```

===== Auction Results =====

Final Average Bid (rounded down): 143

Highest Bid: 160 (Winner: User3)

=====

```

User3 is declared the winner with a bid of £160. The auction is now complete, and the program terminates normally.

Notes for users:

- **Entering Bids:** When entering a bid, a positive integer must be used. It must be equal to or greater than the minimum bid otherwise it will be rejected. Non-integers and empty bids will cause a re-prompt.
- **Termination:** To end the bidding phase early, the letter 'x' can be used when prompted for a user ID. This is not case sensitive so 'X' can be used too.
- **Unique IDs:** Ensure each bidder uses a unique ID, as no other IDs are accepted. If a user ID has been used to place a bid, it will be rejected if there is an attempt to use it again.
- **During Computation:** Due to the competitive cost of homomorphic computations, the process may take a while and will become slower with the more bidders added. During computation there is no input accepted. Do not close the program, it has not frozen, it is completing the computations.
- **Tiebreak Rounds:** If a tie is detected, the program will automatically continue to gather new bids from the users who tied. Bidders should be prepared to input a new bid when prompted. The minimum for the tiebreak is updated to the current highest bid ensuring that no one can lower their bid.
- **Ending Tiebreaks:** Once a tiebreak round produces a clear winner, the program will announce the winner and terminate the program. If ties keep continuing, the tiebreak process will also continue. If a bidder opts to not place another bid then they can enter 'x' (not case sensitive), to essentially drop out of the bidding war.

Troubleshooting:

- If the program terminates unexpectedly or outputs nonsense for the results, it could be due to an internal error or incorrect parameter setup (which shouldn't happen with the provided release build). Please ensure that you ran the release build as configured with the correct parameters
- If the program is too slow, attempt to run on a more powerful machine or limit the number of participants in the auction.
- For all cryptographic or technical errors (Rust panic messages), contact the developer or refer to the code documentation.

9.2. Appendix B: Test Results and Output Logs

Unit Test

Unit test executed against the core homomorphic routines all successfully passed. Below is the output when the unit test is ran:

```
running 5 tests
test list_average ... Test list_average: bids [100, 250, 200, 350], avg = 225
ok
test list_compare ... Test list_compare: bids [100, 400, 250, 150], max = 400,
winners = ["U1"]
ok
test single_value_average ... Test single_value_average: bids [200], avg = 200
ok
test two_different_comparison ... Test two_different_comparison: bids [200,100],
max = 200, winners = ["A"]
ok
test two_equal_comparison ... Test two_equal_comparison: bids [300,300], max = 300,
winners = ["X", "Y"]
ok

test result: ok. 5 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 9.73s
```

Integration Test

Integration test executed against scenarios stated in the testing section of report and all were successfully passed. Below is the output from when the integration test is ran:

```
running 5 tests
test all_bidders_tie ...
```

=====

Starting All Bidders Tie

=====

Initial Bids: [User1: 300, User2: 300, User3: 300, User4: 300]

Tie detected among all users. Starting first tie-break round...

After first tie-break: [User1: 350, User2: 350, User3: 400, User4: 400]

Tie detected between User3 and User4. Starting second tie-break round...

After second tie-break: [User3: 450, User4: 425]

Winner(s): ["User3"]

Highest Bid: 450

Average Bid: 393

✅ Completed All Bidders Tie Scenario

ok

test no_bids ...

=====

Starting No Bids Scenario

=====

No bids placed. Auction correctly terminated.

✅ Completed No Bids Scenario

ok

test no_tie ...

=====

Starting No-Tie Scenario

=====

Bids: [200, 300]

Winner(s): ["User2"]

Highest Bid: 300

Average Bid: 250

✅ Completed No-Tie Scenario

ok

test tie ...

=====

Starting Tie Scenario

=====

Initial Bids: [User1: 100, User2: 250, User3: 250]


Tie detected between User2 and User3. Starting tie-break round...

After tie-break: [User2: 300, User3: 325]

Winner(s): ["User3"]

Highest Bid: 325

Average Bid: 241

 Completed Tie Scenario

ok

test validation_of_minimum_bid ...


=====

Starting Validation of Minimum Bid

=====

Attempted invalid bid: 150

Correctly rejected bid below minimum threshold.

 Completed Validation of Minimum Bid

ok

test result: ok. 5 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;

finished in 12.70s

Performance Test:

Performance test executed on samples of 10, 20 and 50 bids to calculate the runtime of the auction system. The outputs from each performance test were as follows:

10 bids:

```
running 1 test
test performance_test_benchmark ... Running performance test with 10 bidders

=== FHE Benchmark (10 bidders) ===
Key generation      : 1075 ms
Encryption+Store    : 8 ms    (≈0.8 ms / bid)
Max+Average comp    : 11359 ms
Decryption          : 0 ms
Highest bid         : 47243
Average bid         : 23466
ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 12.46s
```

20 bids:

```
running 1 test
test performance_test_benchmark ... Running performance test with 20 bidders

=== FHE Benchmark (20 bidders) ===
Key generation      : 1042 ms
Encryption+Store    : 17 ms   (≈0.8 ms / bid)
Max+Average comp    : 17885 ms
Decryption          : 0 ms
Highest bid         : 49723
Average bid         : 26335
ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 18.96s
```


50 bids:

running 1 test

test performance_test_benchmark ... Running performance test with 50 bidders

=== FHE Benchmark (50 bidders) ===

Key generation : 1006 ms

Encryption+Store : 40 ms (~0.8 ms / bid)

Max+Average comp : 44602 ms

Decryption : 0 ms

Highest bid : 49777

Average bid : 30122

ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 45.66s

9.3. Appendix C: Code Repository and Technical Documentation

The full source code for this project is provided in the attached repository “ZamaFHEBid”. The main implementation points are as follows:

- **Repository Structure:**

- *Cargo.toml* – Rust project file specifying project metadata and dependencies (such as the FHE library).
- *src/main.rs* – Contains the entry-point of the program (key generation, user interaction, bid collection, tiebreak control-flow, and final decryption and display of results)
- *src/auction.rs* – Library module that defines the *Auction* struct and contains all cryptographic functionality (add bid, max bid, average bid, and other helper methods).

- **Building and Running:**

As described in Appendix A, use Cargo to build the system. The code is cross-platform meaning it has no OS-specific calls. Rust’s LLVM backend optimises the heavy arithmetic well in release mode.

- **Code Highlights:**

- **Key Generation:** Handled within *main.rs* via *generate_keys()* from the *tfhe* crate. The client key is retained for encryption and decryption, while the server key is registered via *set_server_key()* to enable server-side operations on encrypted data.
- **Auction Struct:** It is define in *auction.rs* and contains core logic for encrypted bid storage and homomorphic computation, maintaining references to the *ClientKey*, a plaintext minimum bid and a *HashMap<String, FheUint16>* mapping user IDs to encrypted bids.
- **Encryption and Storage:** Performed using *FheUint16::encrypt*, which directly encrypts a 16-bit integer into a TFHE ciphertext. All bids are validated against the minimum bid before encryption.
- **Homomorphic Operations:** *compute_max_encrypted_bid()* and *compute_average_encrypted()*, are both implemented using built0in traits from *tfhe::prelude::**. The max function compares encrypted values using *.max()*, while the average perorms summation using homomorphic 32-bit addition followed by

ciphertext division using a plaintext constant to maintain encryption while computing the result.

- **Tiebreaking Mechanism:** Within `main.rs`, a tiebreaking tool performs plaintext-controlled loops where tied users are prompted to re-bid, iteratively recomputing the encrypted maximum until a winner is discovered. This logic is embedded in the main control flow and integrated with user input prompts.

- **Cryptographic Parameters**

This system utilises the *TFHE-rs* library, which implements fully homomorphic encryption over integers. The scheme implements a levelled TFHE using Learning With Errors (LWE)-based ciphertexts. Parameters are abstracted by the *ConfigBuilder::default()* setup, ensuring a secure key size and enables bootstrapping.

- **Ciphertext Types:** The Code uses *FheUint16* for encrypted bids and *FheUint32* for summations to avoid overflow.
- **Bootstrapping:** Logical operations and comparisons are automatically bootstrapped, which refreshes ciphertexts and prevents noise accumulation.
- **Division:** The TFHE crate supports homomorphic division of ciphertexts using a plaintext constant (i.e. number of bidders), which allows for the average computations without decryption, significantly improving security and privacy.

- **Known Limitations in Code:**

There are multiple practical limitations that are documented within the code and/or were observed during testing:

- **Ciphertext Overflow:**
Bids are encrypted using 16-bit ciphertexts (*FheUint16*), meaning that any bids above 65,535 will wrap around modulo 2^{16} . However, for the purpose of our implementation, this range is acceptable.
- **Console Input:**
The system relies on blocking terminal input which limits its suitability for real-time usage in a networked environment.
- **Single-threaded Design:**
Although this system is functional, the current implementation is single-threaded. All cryptographic operations are executed sequentially. To implement this as a

performance-critical application, parallel comparisons would be the most suitable solution for an increase in runtime efficiency.

- **Reproducibility**

This project is fully reproducible without being reliant on many dependencies. Any machine with Rust and an internet connection can compile and execute this system using the command:

```
cargo run --release
```

The decrypted outputs are always correct and consistent for the same plaintext inputs, confirming semantic security and functional correctness.