cryptography

MDPI

Article

Cryptographically Secure Multiparty Computation and Distributed Auctions Using Homomorphic Encryption

Anunay Kulshrestha *, Akshay Rampuria, Matthew Denton and Ashwin Sreenivas

Department of Computer Science, Stanford University, Stanford, CA 94305, USA; rampuria@stanford.edu (A.R.); mdenton@cs.stanford.edu (M.D.); ashwinsr@stanford.edu (A.S.)

* Correspondence: anunay@cs.stanford.edu

Received: 7 October 2017; Accepted: 7 December 2017; Published: 12 December 2017

Abstract: We introduce a robust framework that allows for cryptographically secure multiparty computations, such as distributed private value auctions. The security is guaranteed by two-sided authentication of all network connections, homomorphically encrypted bids, and the publication of zero-knowledge proofs of every computation. This also allows a non-participant verifier to verify the result of any such computation using only the information broadcasted on the network by each individual bidder. Building on previous work on such systems, we design and implement an extensible framework that puts the described ideas to practice. Apart from the actual implementation of the framework, our biggest contribution is the level of protection we are able to guarantee from attacks described in previous work. In order to provide guidance to users of the library, we analyze the use of zero knowledge proofs in ensuring the correct behavior of each node in a computation. We also describe the usage of the library to perform a private-value distributed auction, as well as the other challenges in implementing the protocol, such as auction registration and certificate distribution. Finally, we provide performance statistics on our implementation of the auction.

Keywords: public key cryptography; multiparty computation; homomorphic encryption; zero-knowledge proofs; distributed systems; secure auctions

1. Introduction

There has been an enormous growth in the frequency of online transactions over the years. Sellers of all kinds offer their products and services to a world of unknown buyers. Buyers, similarly, are an eclectic group of entities who know little about the seller.

An auction, simply, is a set of trading rules for the exchange of a good. Typically, auctions have a predetermined procedures that they follow to determine the winner(s). In most auctions where the identity of other bidders is public, there is a risk that a subset of bidders will form a group and attempt to manipulate the auction result—a practice known as collusion. Collusion is one of the biggest problems in current online auctions. For example, depending on the nature of the auction, bidders could collaborate to shade their bids in a contrived way to maximize shared utility. There is also the possibility of collusion between the auctioneer and the bidders in the case where the auctioneer discloses the private bid of one of the bidders to another bidder, thereby affecting the informed bidder's strategy.

In order to protect the interests of both concerned stakeholders, we propose our platform, which can conduct secure private auctions. Sellers can avoid the auctioneer/seller from colluding with any of the participants because the bids are similarly anonymous and homomorphically encrypted. There has been some theoretical work done on this front, but we attempted to be the first and only open

sourced implementation that is backed by a powerful library that performs the security, distribution, collation, and assimilation pieces.

The library supports the design of special-purpose protocols for computing a subset of general functions. For example, a simple private Hamming distance protocol is presented. In addition, the library can be used for an extension of the first-price auction described below, to $(M + 1)^{st}$ price auctions. This general class of auctions has many attractive theoretical guarantees, but has not seen widespread use due to mistrust of the seller. However, with the security mechanisms described in this paper, correctness of the result can be mathematically guaranteed, all the while preserving the privacy of each bid.

The aim of this paper is to describe the theory and implementation of the library, and provide detailed guidelines on its usage, as well as guidelines on the creation of secure special-purpose protocols. The rest of the paper is organized as follows. Section 2 reviews related work, Section 3 defines mathematical prerequisites, Section 4 describes the proposed library, Sections 5 and 6 discuss auction protocols, Section 7 formulates general guidelines for future work and Section 8 handles auction registration. Finally, Section 9 discusses possible extensions and presents conclusions.

2. Related Work

Secure auctions (that protect the identity of the bidders and their sealed bids) have been studied extensively in recent years [1–3]. However, work on secure auctions has focused on either a trusted third party auctioneer or an auction service to run auctions [1]. Consequently, there has been limited discussion of distributed frameworks for achieving this end [4,5].

Brandt provides the first distributed framework based on multiparty computation that uses homomorphic encryption to run auctions [6,7]. This framework is more extensible than past work and can be used to run any auction that can be represented as a Boolean predicate. However, Dreier et al. identify few security vulnerabilities in this framework [8]. From a theoretical perspective, this paper builds on Brandt's framework and defends against most of these vulnerabilities. From the perspective of implementation, this paper contributes the first—to our knowledge—efficient implementation of an extensible library that can used to run distributed secure auctions, with provable security guarantees.

3. Non-Interactive Zero Knowledge Proofs Using the Fiat–Shamir Heuristic

As in Brandt's work, our framework employs four types of zero knowledge proofs. These proofs are used to verify each step in secure, fully-private protocols as we describe below [6,7].

However, zero knowledge proofs often necessitate that the verifier send a randomly generated "challenge" to the prover. The verifier can be convinced of the proof's validity when the prover sends back a correct response to the challenge.

We use the Fiat–Shamir heuristic [9] to *flatten* the proofs and eliminate the need for random "challenges". This non-interactivity not only significantly cuts down on network traffic and latency, but makes the proofs non-malleable and secure against attacks described by Dreier et al. [8].

We discuss each proof in detail in the subsequent sections. Throughout the proofs, we use a cryptographic hash function (like SHA-256) to emulate the access to a random oracle that the Fiat–Shamir heuristic requires. Typically, the input to the hash functions includes every publicly known value in order to maximize the randomness of the resulting challenge, while still enabling the verifier to calculate the challenge. The correctness of this approach has been proven by Bellare et al. in 1993 [10].

3.1. Preliminaries

Our framework uses the El Gamal Cryptosystem [11], which is probabilistic and homomorphic. We choose two primes p and q so that there exists $k > 1$ such that:

$$p = qk + 1.$$

$\mathbb{G}_q$ denotes the unique multiplicative subgroup of order q in $\mathbb{Z}_p$. All computations in the remainder of this paper are modulo p unless otherwise noted. For a given participant, the private key is $x \in \mathbb{Z}_q$, the public key is $y = g^x$, where $g \in \mathbb{G}_q$ is an arbitrary, publicly known element. A message $m \in \mathbb{G}_q$ is encrypted by computing the ciphertext tuple:

$$(\alpha, \beta) = (my^r, g^r),$$

where $r \in \mathbb{Z}_q$ is an arbitrary random number, chosen by the encrypter. A message is decrypted by computing:

$$\frac{\alpha}{\beta^x} = \frac{my^r}{(g^r)^x} = m.$$

El Gamal is homomorphic as the component-wise product of two ciphertexts

$$(\alpha\alpha', \beta\beta') = (mm'y^{r+r'}, g^{r+r'})$$

represents an encryption of the plaintext's product mm'.

It has been shown that El Gamal is semantically secure, i.e., it is computationally infeasible to distinguish between the encryptions of any two given messages, if the decisional Diffie–Hellman problem is intractable [12]. We will use functions $E(\cdot)$ and $D(\cdot)$ to denote the encryption and the decryption of plain- and ciphertexts, respectively.

3.2. Distributed El Gamal Encryption

A further useful property of El Gamal is its ability to distribute encryption and decryption across multiple nodes, in which no single node or group of nodes can decrypt without cooperation from every node.

Each node a generates a sufficiently random private key x_a, which will be included in the global private key. Recall that $g \in \mathbb{G}_q$ is an arbitrary, publicly known element. Each node publishes the partial public key $y_a = g^{x_a}$. The global public key is defined as:

$$y = \prod_{i=1}^{n} y_a,$$

where n is the number of participants. The global public key can be computed by each of the participants autonomously. Notice that:

$$y = g^{\sum_{i=1}^{n} x_a},$$

which produces a global private key x:

$$x = \sum_{i=1}^{n} x_a,$$

though this is unknown to any single node, as long as the nodes are not all sharing their individual private keys, due to the intractability of calculating the discrete logarithm of $y = g^x$.

Then, to decrypt a message (my^r, g^r), each participant publishes:

$$\phi_a = g^{rx_a}.$$

Then, each participant may independently calculate:

$$\prod_{i=1}^{n} \phi_a = g^{r \sum_{i=1}^{n} x_a} = g^{xr} = y^r.$$

Furthermore, the plaintext can be produced by calculating $\frac{my^r}{y^r}$. Due to the nature of modular exponentiation, if a single bidder Bob does not exponentiate with his private key, no information can be obtained about the plaintext. Thus, each node must cooperate in the joint decryption, and, if a single node is unhappy with the behavior of the other nodes, it may refuse to decrypt the result.

3.3. Proof of Knowledge of a Discrete Logarithm

This protocol is due to Schnorr [13]. Alice (A) and Bob (B) know y_A and g, but only Alice knows x_A, so that $y_A = g^{x_A}$. In the context of our framework, (x_A, y_A) represents Alice's private-public key pair used to guarantee her identity. Naturally, y_A is public but x_A is not. Note that g is also public and common to all participants.

Alice chooses z at random and

$$t_A = g^z.$$

Then, she computes

$$c_A = H(g \oplus y_A \oplus t_A),$$

where $a \oplus b$ represents the string concatenation of (possibly integers) a and b and $H(a)$ is the 256-bit Secure Hash Algorithm (SHA). Alice also computes

$$r_A = z - cx_A.$$

The proof (t_A, r_A) is published to all other nodes. Every node independently verifies it by computing

$$c_A = H(g \oplus y_A \oplus t_A),$$

and checking if

$$t = g^{r_A} {y_A}^{c_A}.$$

3.4. Proof of the Equality of Two Discrete Logarithms

When executing the previous protocol in parallel, the equality of two discrete logarithms can be proven.

Alice (A) and Bob (B) know y_1, y_2, but only Alice knows x_A, so that $y_1 = g_1^{x_A}$ and $y_2 = g_2^{x_A}$. Note that g_1, g_2 are public and commonly known to all participants.

Alice chooses z at random and

$$\begin{aligned} t_1 &= {g_1}^{z_1}, \\ t_2 &= {g_2}^{z_2}. \end{aligned}$$

Then, she computes

$$c_A = H(g_1 \oplus g_2 \oplus y_1 \oplus y_2 \oplus t_1 \oplus t_2).$$

Alice also computes

$$r_A = z - cx_A.$$

The proof (t_1, t_2, r_A) is published to all other nodes. Every node independently verifies it by computing

$$c_A = H(g_1 \oplus g_2 \oplus y_1 \oplus y_2 \oplus t_1 \oplus t_2),$$

and checking if

$$\begin{aligned} t_1 &= g_1^{r_A} {y_1}^{c_A}, \\ t_2 &= g_2^{r_A} {y_2}^{c_A}. \end{aligned}$$

3.5. Proof That an Encrypted Value Is One out of Two Values

The following protocol was proposed by Cramer et al. [14].

Alice proves that an El Gamal encrypted value $(\alpha, \beta) = (my^r, g^r)$ either decrypts to 1 or to a fixed value $z \in \mathbb{G}_q$ without revealing which is the case; in other words, it is shown that $m \in \{1, z\}$. A proof is a tuple $((\alpha, \beta)a_1, a_2, b_1, b_2, d_1, d_2, r_1)$.

If $m = 1$, Alice chooses $r_1, d_1, w \in \mathbb{Z}_q$ at random. She computes:

$$\begin{aligned} a_1 &= g^{r_1}\beta^{d_1}, \\ b_1 &= y^{r_1}(\frac{\alpha}{z})^{d_1}, \\ a_2 &= g^{w}, \\ b_2 &= y^{w}, \\ c &= H(a_1 \oplus a_2 \oplus b_1 \oplus b_2) \pmod q, \\ d_2 &= c - d_1 \pmod q, \\ r_2 &= w - rd_2 \pmod q. \end{aligned}$$

Otherwise, if $m = z$, then Alice chooses $r_2, d_2, w \in \mathbb{Z}_q$ at random. She computes:

$$\begin{aligned} a_1 &= g^{w}, \\ b_1 &= y^{w}, \\ a_2 &= g^{r_2}\beta^{d_2}, \\ b_2 &= y^{r_2}\beta^{d_2}, \\ c &= H(a_1 \oplus a_2 \oplus b_1 \oplus b_2) \pmod q, \\ d_1 &= c - d_2 \pmod q, \\ r_1 &= w - rd_1 \pmod q. \end{aligned}$$

The proof $((\alpha, \beta)a_1, a_2, b_1, b_2, d_1, d_2, r_1)$ is published to all other nodes. Every node independently verifies it by computing:

$$c = H(a_1 \oplus a_2 \oplus b_1 \oplus b_2) \pmod q,$$

and checking if:

$$\begin{aligned} a_1 &= g^{r_1}\beta^{d_1}, \\ b_1 &= y^{r_1}(\tfrac{a}{z})^{d_1}, \\ a_2 &= g^{r_2}\beta^{d_2}, \\ a_2 &= g^{r_2}\beta^{d_2}. \end{aligned}$$

3.6. Verifiable Secret Shuffle of Ciphertexts

A shuffle is a rearrangement and reencryption of input ciphertexts. While shuffling is easy, it is hard for an outsider to verify that a shuffle has been performed correctly. By proving such a shuffle correct, a party can verifiably rearrange a vector of ciphertexts without revealing the applied permutation. We proved the secret shuffle of ciphertexts using Groth's protocol [15]. Groth's protocol includes commitments over $\mathbb{Z}_q$ and interactive challenges. In our implementation, we flatten Groth's

proof using the Fiat–Shamir heuristic to make them robust against attacks like those described by Dreier et al. [8].

3.7. Counting Boolean Disjunctions of Literals

We will now show how the primitives defined in the previous section can be used to evaluate simple Boolean expressions [6].

Consider n parties whose inputs are bitstrings b_i of length k. We define $E[b]$ to be an (El Gamal) encryption of bit b.

If $E[0]$ decrypts to 1 and $E[1]$ decrypts to any other number in $\mathbb{G}_q$:

$$D[E[b]] \in \begin{cases} \{1\}, & \text{if } b = 0, \\ \mathbb{G}_q \backslash \{1\}, & \text{otherwise.} \end{cases}$$

Let Y be an arbitrary, publicly known, fixed element of $\mathbb{G}_q \setminus \{1\}$. Before the actual protocol starts, each agent publishes encryptions of his individual input bits so that

$$E[b_{ij}] = \begin{cases} E(1), & \text{if } b_{ij} = 0, \\ E(Y), & \text{otherwise.} \end{cases}$$

The correctness of inputs can be efficiently proven by showing that each ciphertext either decrypts to 1 or to Y without revealing which case holds (Section 3.4). Based on this representation, we can count the number of true Boolean disjunctions of literals and pairwise exclusive disjunctions of literals by computing

$$f(b_1, b_2, \ldots, b_n) = \#\Big(\bigvee_r L_r \vee \bigvee_{s,t} (L_s \oplus L_t) \Big),$$

where $L_r, L_s, L_t \in \{b_{ij} \mid 1 \leq i \leq n, 1 \leq j \leq k\}$ and # is a count operator that counts the number of true expressions. The individual operators can be implemented as follows.

3.8. Negations

The Boolean negation of input bits can be computed by dividing Y by the input bit's encryption:

$$E[\neg b_{ij}] = \frac{Y}{E[b_{ij}].}$$

3.9. Disjunctions

The product of ciphertexts yields the logical *OR* of the corresponding plaintext bits. Please refer to [6] for an in-depth analysis.

$$E\Big[\bigvee_r L_r\Big] = \prod_r E[L_r].$$

3.10. Exclusive Disjunctions

The exclusive OR of pairs of literals can be computed by dividing the encryptions of these bits. Suppose $L_s, L_t \in \{b_{ij} \mid 1 \leq i \leq n, 1 \leq j \leq k\}$; then,

$$E[L_s \oplus L_t] = \frac{E[L_s]}{E[L_t].}$$

3.11. Count Operator

We can count the number of true bits in a vector of encrypted bits by consecutively letting each party verifiably shuffle its vector of bits, thus effectively emulating a mix-net. After exponentiating each component with a jointly created random number and decrypting all components, the number of true bits is exactly the number of components not equal to 1.

4. Library

Using Brandt's framework [6], we designed and implemented a compact yet extensible library that allows a user to run any private multiparty computation protocol or auction building on the cryptographic primitives described in the preceding section. See Figure 1 for a summary of how a protocol, using these building blocks, is generally constructed. The library fully implements:

- Networking and establishment of secure connections (through SSL, with both ends authenticated with certificates)
- Broadcasting of data
- Serialization/deserialization
- Multithreading of computations
- Synchronization
- Efficient generation and verification of zero knowledge proofs

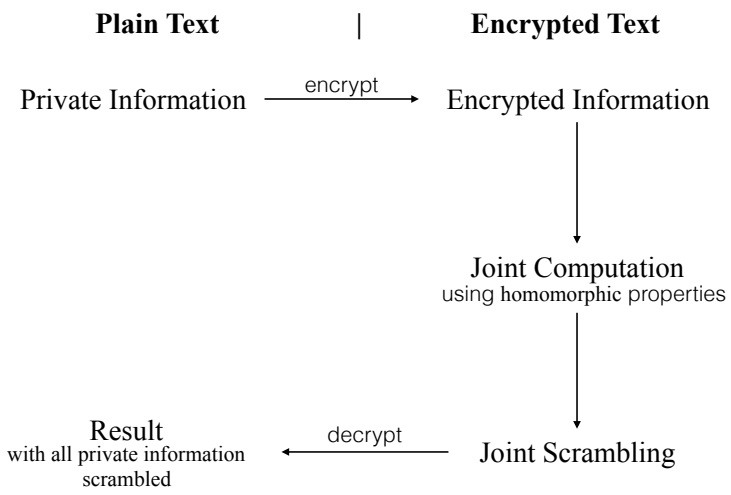

Figure 1. A pictographic representation of the flow of a typical multi-party computation protocol.

4.1. Frameworks Used

Underlying network code uses Google's gRPC and protobuf implementations for establishing connections and securely and reliably distributing data. Works such as Brandt's tend to underestimate the amount of data sent over the network due to the necessity of attaching metadata to the network requests. Our implementations aims to put a concrete bound on the amount of data sent over the network in a fully general implementation of the framework.

Protobuf is a library written by Google used to serialize data into self-contained units to send over the network. The advantage of protobuf is its compact representation of data. Other serialization protocols, such as JSON (JavaScript Object Notation), aim to be human readable and therefore take much more space than is necessary to transport the data.

gRPC is also a library written by Google. It provides reliable transport, integration with protobuf, and a robust implementation of SSL. This includes supports for adding root certificates, as well as server and client authentication. Using these frameworks allows us to take advantage of improvements in performance and security with no updates to our code. gRPC is also used because it can be leveraged by users of our library. Users can take advantage of the improvements in gRPC, as well as the documentation and ease of use.

4.2. Computational Efficiency of the Library

Our library also aims to be computationally efficient. Modular arithmetic is of course always performed in the most efficient way possible. In addition, each connection is serviced in a separate thread, and publishing of data is also serviced in a separate thread. Synchronization between threads is handled automatically by the library. Checking of zero knowledge proofs is done automatically in parallel. General per-round computations are under control of the user, but these can be multithreaded using Golang's simple "goroutines" and synchronization mechanisms.

Our library does not aim to be space efficient, but stores only a negligible amount of data that does not go over the network. Therefore, the network will become a bottleneck far before the available memory space of each node. Figure 2 provides some proof of this: the total memory consumption by an application running Brandt's First-Price Auction Protocol [6] using our library is linear in the number of participants.

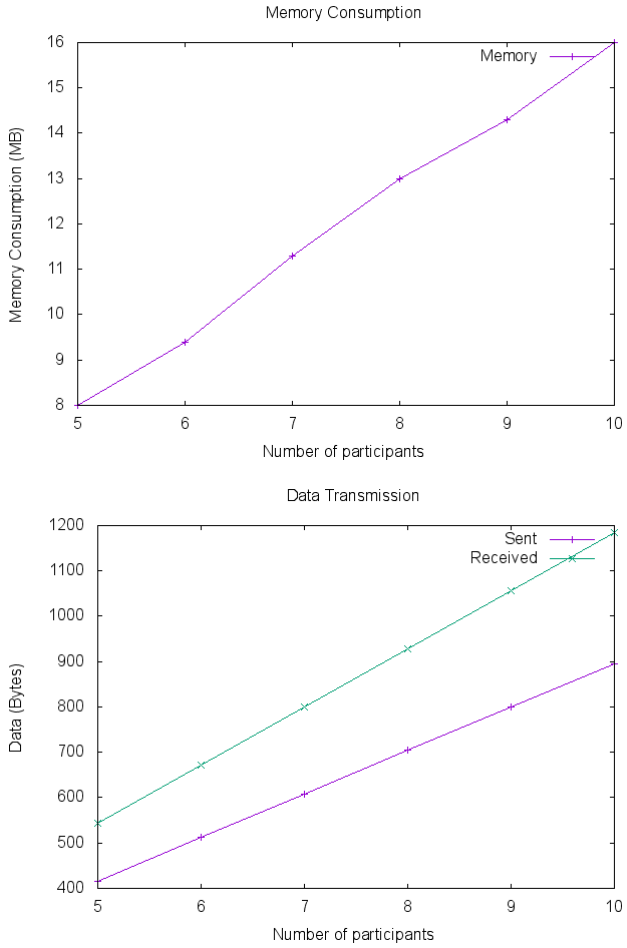

Figure 2. Performance graphs for our implementation of Brandt's First-Price Auction Protocol.

Each client maintains a connection to each of the other clients. Modern operating systems are perfectly capable of maintaining many open connections at once. As there is no state kept within the network itself, it is not a problem for the network that there are $O(n^2)$ connections. Each client's operating systems simply keeps a small amount of state for each other client, which is essentially negligible on modern systems.

Unfortunately, instead of broadcasting each round's data once, each client sends the round's computations to each other client, which does in fact cause more data flow through the network. Alternately, this could be done very efficiently with network-supported multicast, but, unfortunately, there is a lack of multicast support in today's Internet backbone. Another possibility is the use of an append-only bulletin board at a rendezvous point; however, this not only introduces a bottleneck but requires trust of a rendezvous point. We could use the trusted third party, as described below in the section on auction registration. Alternately, we could maintain a global network of peers and select a rendezvous node randomly using a distributed hash table, and use the randomness of rendezvous point to minimize the chance of collusion. See the "Registering for an Auction" and "Extensions" sections for further information.

4.3. Design of the Library

For every round of the protocol, the user must provide three function pointers. They point to three distinct functions that define the round: *compute*, *check* and *receive*. Each round may be specified at the beginning of the program, or added dynamically based on the results of the protocol.

The *compute* function computes the variables necessary for broadcasting and checking of zero knowledge proofs. It is run sequentially and happens after the previous round ends, though the user may use threading within the *compute* function to parallelize difficult computations. The *compute* function may return a populated user-defined protobuf structure, and the library will automatically publish it to the other participants in parallel. After pre-computation, the library will begin to process data received from other participants. Note that, due to inconsistencies in timing between the participants, a node may receive data from another participant that is from a future round, and its accompanying zero knowledge proofs cannot be checked yet if computation has not yet finished. The library automatically handles the situation and delays checking of the zero knowledge proofs until the computation step for the same round has been performed.

The *check* function checks the zero knowledge proof for each individual participant. These proofs will be run in separate threads in parallel in order to maximize efficiency. The *check* function is guaranteed not to be run until the *compute* function for the same round has finished. This sequential requirement is due to the fact that, in order to check some zero knowledge proofs from other participants, a node must have precomputed data.

When data has been received from every other participant, and each node's data has been verified with the *check* function, the *receive* function is called with all the data. The *receive* function may save all necessary data in the program state, and may discard the zero knowledge proofs now that they have been verified and are no longer necessary. The reason for separate *check* and *receive* functions is the utility of parallelizing the *check* function, which therefore may not modify state. The *receive* function is not called in parallel with any other function.

4.4. Additional Library Functionality

The library also contains efficient implementations of the zero knowledge proofs mentioned above. This includes both generation and checking of the proofs. These can be employed easily by users of the library.

It also provides extensive helper functions for deserialization and processing of large integers.

Finally, it provides efficient key generation, as well as IDs and connections between the clients, which can be utilized for other communication between clients, should the protocol dictate. For example, the first price auction protocol, detailed elsewhere in this paper, requires each bidder to

communicate their computed data to the seller instead of the other bidders, so that bidders cannot learn the result of the auction early and exit before communicating their final computed data.

5. Millionaires' Protocol

Millionaires' Problem was first defined by Andrew Yao in 1982 [16]. More generally, given two numbers a and b, the problem asks to solve the inequality $a \leq b$ without revealing the value of a or b.

In Yao's formulation, the problem involves two millionaires Alice and Bob, who wish to discern who is richer but do not wish to disclose their own wealth. This is effectively the greater than function.

Note that if a and b are binary representations of two numbers and a_i, b_i are the ith digits from the right, the greater than function can be reformulated in the following way:

$$[a > b] \iff \bigvee_{i=1}^{k} \left(a_j \wedge \neg b_j \wedge \bigwedge_{d=j+1}^{k} (\neg(a_d \oplus b_d)) \right).$$

Applying De Morgan's law and leveraging the predicates described earlier, this could be computed as a product and exponentiation of ciphertexts. Brandt describes such a protocol in his original paper [6].

We implemented this protocol using our library, and broke it up into various rounds of *compute*, *check*, and *receive*. Let us now look into an auction protocol more closely and examine how we could use the library to implement it seamlessly.

6. First-Price Auction Protocol

The following is an implementation of Brandt's first price auction protocol, detailed in [7]. This protocol serves as an example of how the library may be leveraged to write custom protocols, and how existing protocols can be separated into distinct *compute*, *check*, and *receive* functions.

Note that, in each of the following *compute* function descriptions, the word "broadcast" refers to the process of inserting the specified data into a custom written protobuf structure. At the conclusion of the *compute* function, the protobuf structure is optionally passed to the library, which will automatically broadcast the structure to each of the other clients (over the network).

The structures are defined by the user of the library, and therefore there is no specific format imposed by the library. The only requirement is that the structure is defined using protobuf. As such, in our implementation of the following protocol, we define a custom protobuf structure for each round. These structures can be easily inferred from the description of the compute function; for more details (see the code in the publicly available Git repository [17]).

One of the features of the first-price protocol is that it only allows you to bid on one of $b_1, b_2, ..., b_k$ discrete values. Note that the bid vectors b_a looks like the following:

$$b_a = \begin{bmatrix} b_{ak} \\ . \\ . \\ b_{aj} = 1 \\ . \\ . \\ b_{a1} \end{bmatrix}. \tag{1}$$

Here, there is only one element—the actual bid—that is 1. The rest of the element is 0. The position of the 1 indicates which of the k bids bidder a actually bids on. There could be a workaround, like a hierarchical bidding structure with multiple rounds to allow for any discrete value, but it was not our focus.

Note that all computations, if not specified, are performed modulo p, one of the large primes used in El Gamal encryption (see previous sections on the El Gamal cryptography scheme).

6.1. Prologue: Generate Public Key

Compute:

- Choose private key $x_{+a} \in \mathbb{Z}_q$ randomly.
- Save private key x_{+a} in state for use in final decryption round.
- Broadcast $y_{\times a} = g^{x_{+a}}$ along with a zero-knowledge proof of knowledge of $y_{\times a}$'s discrete logarithm.
- Save public key $y_{\times a}$ in state for use in calculating global public key.

Check:

- Check the correct deserialization of data and the legality of the proof and data values (e.g., $y_{\times a} \in Z_p \setminus \{0\}$).
- Check all the proofs we have received from the other bidders according to the $y_{\times i}$'s. i.e., check if each bidder has knowledge of the discrete logarithm of their broadcasted public key.

Receive:

- On obtaining all the public keys, $y_{\times i}$ for each player, store the public keys in state.
- Compute and store the global public key $y = \prod_{i=1}^{n} y_{\times i}$.

6.2. Round 1: Encrypt Bid

Compute:

- Set $b_{aj} = Y$ if $j = b_a$ else $b_{aj} = 0$.
- Broadcast $\alpha_{aj} = b_{aj} y^{r_{aj}}$ and $\beta_{aj} = g^{r_{aj}}$ for each j, where each r_{ij} is a private, randomly generated number.
- Save each α_{aj} and β_{aj} in state for future use.
- Broadcast zero-knowledge proofs for encrypted value to be one of two values $\forall j : \log_g \beta_{aj}$ equals $\log_y \alpha_{aj}$ or $\log_y \left(\frac{\alpha_{aj}}{Y}\right)$.
- Broadcast zero-knowledge proof for discrete log equality for $\log_y \left(\frac{\prod_{j=1}^{k} \alpha_{aj}}{Y}\right) = \log_g \left(\prod_{j=1}^{k} \alpha_{aj}\right)$.

Check:

- Check the correct deserialization of data and the legality of the proof and data values (e.g., $\alpha_{aj} \in \mathbb{Z}_p \setminus \{0\}$).
- Verify all the zero-knowledge proofs received for encrypted value being one of two and for discrete log equality.

Receive:

- Save all the received α_{aj} and β_{aj} into state so that it can be used for subsequent computations.

6.3. Round 2: Compute Outcome

Compute:

- For each i and j compute and broadcast:

$$\gamma_{ij}^{\times a} = \left(\left(\prod_{h=1}^{n} \prod_{d=j+1}^{k} \alpha_{hd}\right)\left(\prod_{d=1}^{j-1} \alpha_{id}\right)\left(\prod_{h=1}^{i-1} \alpha_{hj}\right)\right)^{m_{ij}^{+a}},$$

$$\delta_{ij}^{\times a} = \left(\left(\prod_{h=1}^{n} \prod_{d=j+1}^{k} \beta_{hd}\right)\left(\prod_{d=1}^{j-1} \beta_{id}\right)\left(\prod_{h=1}^{i-1} \beta_{hj}\right)\right)^{m_{ij}^{+a}}.$$

- In addition, broadcast a proof for discrete logarithm equality to prove that the respective $\gamma_{ij}^{\times a}$ and $\delta_{ij}^{\times a}$ are exponentiated with the same m_{ij}^{+a}.

- Save each $\gamma_{ij}^{\times a}$ and $\delta_{ij}^{\times a}$ into a state for use in verifying the other bidders' zero knowledge proofs.

Check:

- Check the correct deserialization of data and the legality of the proof and data values (e.g., $\gamma_{ij}^{\times a} \in \mathbb{Z}_p \setminus \{0\}$).
- Check proofs of correctness for all the γs and δs that were received from the other bidders; i.e., check that each of the $\gamma_{ij}^{\times a}$ and $\delta_{ij}^{\times a}$ calculated in the *compute* function were exponentiated by a value known to the sender.

Receive:

- Save all the γs and δs that were received.

6.4. Round 3: Joint Decryption

Compute:

- Send $\varphi_{ij}^{\times a} = (\prod_{h=1}^{n} \delta_{ij}^{\times h})^{x_{+a}}$ directly to the seller along with a proof of discrete log equality (this unicast directive is signalled to the library, rather than done manually by the *compute* function).
- Save each $\varphi_{ij}^{\times a}$ into state in order to decrypt in the epilogue.

Check:

- Check the correct deserialization of data and the legality of the proof and data values (e.g., $\varphi_{ij}^{\times a} \in \mathbb{Z}_p \setminus \{0\}$).
- Verify the zero-knowledge proofs of discrete log equality received from all the bidders.

Sellers Receive:

- Save all the received $\varphi_{ij}^{\times a}$ into seller's state so that it can be used for subsequent computations.
- Relay every $\varphi_{ij}^{\times a}$ and its corresponding proof received to all the bidders.

Bidders Receive:

- Save all the received $\varphi_{ij}^{\times a}$ into state so that it can be used for subsequent computations.

6.5. Epilogue: Determine Winner

- On obtaining all δs from round 3, for all j, compute:

$$v_{aj} = \frac{\prod_{i=1}^{n} \gamma_{aj}^{\times i}}{\prod_{i=1}^{n} \delta_{aj}^{\times i}}.$$

If $v_{aw} = 1$ for any w, then bidder a is the winner of the auction and p_w is the selling price. Note that this is the only piece of information available to all the participants in the auction, including the seller.

7. General Guidelines for ZKPs through Hamming Distance Protocol

Each round of a protocol must be verified using zero knowledge proofs; otherwise, a malicious bidder could manipulate the computation in order to manipulate the encryption or the result.

As general guidelines on using zero knowledge proofs to verify individual steps, we present a simple Hamming distance protocol as an example, in which two participants calculate the Hamming distance between two bitstrings, without revealing the actual bitstrings. This can serve many uses. As a simple example, consider a dating service in which each bit of the bitstring represents a preference

for a certain activity. Users of the service can compare their "compatibility" with another user by calculating the Hamming distance between these two strings. Furthermore, the users can calculate the Hamming distance without revealing their preferences to the other user, nor the service itself. This may be for reasons of privacy or an aversion to sharing personal data with a third-party service. Of course, Hamming distance between bitstrings is not an especially secret metric and should only be performed once to preserve privacy; however, the protocol below serves as a sufficiently simple example.

For more detailed protocol information, see the specification of the millionaire's protocol in Brandt's paper [6], which is similar to the following protocol.

7.1. Key Distribution

Bidders Publish:

$$y = g^x.$$

g and y are publicly known, but x must be kept secret by the bidder. However, a malicious bidder may publish a specially calculated value as the public key in order to manipulate the calculations. Because calculating the discrete logarithm of this specially computed value is computationally intractable for the attacker, requiring a knowledge of the discrete logarithm of y provides a significant hindrance to attackers. For example, if a malicious bidder Mallory (with ID ω) received the other public keys $y_1, ..., y_{\omega-1}, y_{\omega+1}, ..., y_n$, and then calculated his public key as

$$y_\omega = \left(\prod_{i \neq \omega} y_i \right)^{-1}.$$

Then, if no proof of knowledge is required for the discrete logarithm of y_ω, the public key will be calculated as

$$y = \left(\prod_{i \neq \omega} y_i \right) y_\omega = \left(\prod_{i \neq \omega} y_i \right) \left(\prod_{i \neq \omega} y_i \right)^{-1} = 1,$$

and then all calculations will be performed under El Gamal encryption with a public key of 1; i.e., completely in the clear. Other, more sophisticated and less detectable attacks are also possible. Thus, requiring the knowledge of the public key's discrete logarithm greatly constrains the possibility of manipulation of the computation.

7.2. Publication of Encrypted Inputs

Each bidder i, for each bit of their bitstring b_{ij}, publishes the El Gamal ciphertext:

$$(\alpha_{ij}, \beta_{ij}) = (g^{b_{ij}} y^{r_{ij}}, g^{r_{ij}}),$$

where r_{ij} is a private randomly generated value, and must publish a proof (for each j) that the encrypted value is either g^0 or g^1. It may seem sufficient to publish knowledge of the discrete logarithm of βij. Unfortunately, it is not. It is possible that, for encrypting special values, it is possible to recover the other participant's entire bitstring. It is advisable to always use the strongest zero knowledge proof available.

7.3. Computation and First Scrambling

Bidders calculate for each j:

$$\gamma_j = \frac{\alpha_{1j}}{\alpha_{2j}},$$

$$\delta_j = \frac{\beta_{1j}}{\beta_{2j}}.$$

After this, if decryption were to take place, the bits would decrypt to their exclusive disjunction. However, we wish to hide the exact bits and their ordering; otherwise, each bidder could calculate the other's bitstring at will. Thus, Bidder 1 publishes a shuffle of each of the values, along with a proof of the secret shuffling. Bidder 2 also calculated the original values and can use these as an input to the verifier, along with the zero knowledge proof, in order to verify the shuffle. Notice that the proof of the shuffle serves more of a purpose than just verifying that Bidder 1 performed a shuffle of some values. Because Bidder 2 also calculated the values to be shuffled, verifying the shuffle also verifies that Bidder 1 performed the calculation correctly and without malicious modifications.

7.4. Second Scrambling

Bidder 2 now must scramble the shuffled values received from Bidder 1: in general purpose computations, all n bidders must shuffle the computed values. If there were some subset of shufflers, the shufflers may collude and have knowledge of the final permutation. Thus, if any bidder is not allowed to shuffle values, that bidder may refuse to participate in decryption, causing the failure of the entire decryption process. For the purposes of our Hamming distance protocol, Bidder 2 also must publish proof of the secret shuffle of Bidder 1's values.

7.5. Randomizing Output

Information may still be recovered from the final decrypted values, even despite the secret shuffling above. Thus, we perform a final scrambling step: joint exponentiation by a random number. Each Bidder i, for each j, publishes:

$$\gamma_j^i = \gamma_j^{m_{ij}},$$
$$\delta_j^i = \delta_j^{m_{ij}},$$

where m_{ij} is a private randomly generated number. Thus, also encrypted values that are not g^0 are transformed into random numbers. A proof of equality of the discrete logarithms of γ_j^i and δ_j^i is also published. This prevents publishing of maliciously generated values. In addition, when Bidder 2 is checking that γ_j^1 and δ_j^1 have the same discrete logarithm, Bidder 2 can also be ensured that Bidder 1 used the values as randomly shuffled by Bidder 2.

For a fascinating attack on Brandt's first price auction protocol, using manipulation of the randomly generated exponents and a man-in-the middle attack on malleable zero knowledge proofs, see [8].

7.6. Decryption

Each Bidder i, for each j, publishes:

$$\phi_j^i = (\delta_j^1 \delta_j^2)^{x_i}.$$

In addition, for each j, a bidder publishes a proof that ϕ_j^i has the same discrete logarithm as y_i; i.e., proves that the decryption is being done with the same private key used to generate the public key. Then, each bidder may calculate the final decryption:

$$v_j = \frac{\gamma_j^1 \gamma_j^2}{\phi_j^1 \phi_j^2}.$$

The number of values not equal to 1 is the final Hamming distance.

If instead the bidders only published proofs that the ϕ values have the same discrete logarithm, the calculation may still be manipulated. For example, one bidder can learn the result of the Hamming distance protocol before sending his final ϕ values. Then, the bidder can publish his final ϕ values with a random exponent not equal to his private key. The other, non-malicious bidder can verify that the proofs are correct, but all of the decrypted values will be garbage values. As discussed below,

the non-malicious bidder will then believe, because the proofs checked out, that the Hamming distance was k, the length of the bitstring. Thus, a malicious bidder would be able to convince the non-malicious bidder of an incorrect final result, while learning the correct result himself.

To conclude this section, it should be noted that the strongest available zero knowledge proofs should be utilized. In addition, zero knowledge proofs of discrete logarithms can be used to verify not only the exponents, but the bases. Finally, in exponentiations involving the private key, verifiers must be careful to check that the exponents are the same as the private key, and not just the same as each other.

8. Registering for an Auction

Since our library is designed to support distributed auctions, we solve the problem of host discovery by having all participants register for an auction before it begins. Having participants register for an auction solves two problems:

1. All clients in the auction know who all the other clients in the auction are (or at least the ones that they have to connect to).
2. All clients are provided with a method by which they can authenticate connections to other clients (to prevent man-in-the-middle attacks).

The registration process for the auction occurs in three steps—auction creation, registration, and auction file creation. A single trusted third party (TTP) server handles these three steps. However, note that each of the following three steps must take place over a secure channel such as SSL (Secure Sockets Layer). This makes the TTP's Certificate Authority (CA), who provided signing of the TTP's certificate, a joint root of trust.

8.1. Auction Creation

In our implementation, a simple HTTP GET request by a seller to the server can be used to create a new auction. At this stage, alternative implementations could specify the kind of auction format that will be used, description of the good being auctioned, minimum bids, authentication of the seller, etc.

8.2. Registration

At this stage, bidders register for an auction that has been created. Our implementation allows any potential bidder to register for any auction, for simplicity, but this can easily be extended to restrict access in various ways such as Internet Protocol (IP) whitelists and blacklists or login authentication for example. Upon registration, each bidder is given a certificate that is signed by the trusted third party (who acts as a CA). This is done to allow bidders to use these certificates to verify the authenticity of other bidders during the distributed phase of the auction. Now, at the start of distributed computation, both participants mutually verify the authenticity of the other, thereby preventing man-in-the-middle attacks and hence solving problem #2 that we introduced earlier.

8.3. Auction File Creation

By this third, and final, stage, all interested bidders have already registered for the auction and received their certificates. At the very start of the auction, all participants reach out to the TTP server one last time and request for the auction file. The auction file is custom created for each client and acts analogously to a torrent file. It includes information about the other participants in the auction that this bidder should connect to. This solves problem #1 that we introduced earlier—clients now know whom they should connect to. In our implementation, the auction file contains the addresses of the other hosts in the auction, the seller's address, and the ID given to each particular participant. This auction file can potentially be extended to include the format of bids to be submitted, details of various rounds of the auction, etc.

The certificates and the auction file together solve both the problems we introduced at the start of this section; with both, participants are ready to begin bidding!

8.4. Attacks by the TTP

Note that the TTP is the single point of failure in our protocol. Because the TTP in our implementation distributes both the certificates and IPs of the other clients, the TTP is capable of being a man-in-the middle for every computation. Dreier et al. [8] describe an attack in which a man-in-the-middle can pose as each of the bidders in an auction by distributing attacker-owned IP addresses and certificates. The fake bidders can bid the lowest valuation in the price vector, and thus the computation will take place and the proofs will check out, but the attacker learns the bidder's actual bid.

Their attack can be extended when trusting the TTP: a malicious TTP can create a fake auction for each of the n real bidders. The TTP also distributes a fake certificate and IP address for the seller (as a man-in-the-middle) on the last step of the auction. Each fake auction will take place with n fake bidders and one real bidder. The fake auction will proceed as before, with each fake bidder bidding 0, allowing the attacker to discern the real bidder's bid. However, on the final step, one of the fake bidders will produce an incorrect zero knowledge proof, causing the real bidder to remove that fake IP from the auction, as specified in Brandt's original protocol [7]. There will remain $n-1$ fake bidders in the auction. Once every fake auction has reached this point, the real auction can now take place, with the malicious TTP routing the traffic from each real bidder to the other $n-1$ real bidders, through the fake IP addresses that have been distributed to each real bidder. As such, the auction will conclude correctly, and nothing will appear amiss to any of the real bidders, but the TTP will have discerned all the bids. With the malicious TTP colluding with the seller or bidders, matters could be even worse.

Trusting a TTP is normally fine for most purposes. For example, another distributed application, BitTorrent relies entirely on trusted third parties: the sites that distribute Torrent files. Users must trust the site entirely in order to download a Torrent file, as the Torrent file completely specifies the user's desired file. The Torrent file may specify another file that contains a hidden virus such as ransom-ware.

In addition, the trusted third party can take other forms. For example, certificates and IP addresses can be posted on public messaging boards such as Reddit or Facebook. Large companies such as these would have little interest in the difficult task of modifying the certificates, due to the amount of data they already have through other means, as well as the difficulty of identifying auctions and modifying the certificates without being detected.

However, if bidders do not want to trust an online third party service, there are more solutions to this problem:

- We can require the TTP to publicly expose the certificates and IP addresses before the auction starts. In this way, each bidder can verify (by connecting through a separate, anonymous IP address) that his/her certificate is correctly returned. Each bidder can then use these publicly exposed certificates to perform the auction protocol.
- Out-of-band sharing of certificates. This entirely solves the problem of authenticity and bid-privacy, but raises issues of anonymity. This could be applicable for auctions involving well-known bidders and well-known sellers. For example, for use in government contracts: well-known companies may want a secure, private auction because they do not trust the government's interference and do not want to reveal their valuations to other competitors.
- Multiple TTPs. Users can register with multiple TTPs, and, as long as all the TTP's lists of IPs and certificates match, the auction can proceed. This unfortunately breaks the advantage of Brandt's protocol in that the auction is safe from any number of colluding bidders (except n), as we require a majority of the TTPs to not collude.
- Clients randomly select a a third party node as the TTP. This can be implemented by maintaining a global network of potential sellers and bidders, and using a distributed hash table (DHT) to find auctions. For example, more recent BitTorrent implementations can utilize a DHT in order

to find peers for downloading a file, by having other peers place their peer information on a node selected by a hash of the Torrent ID, and finding the information using the DHT. When an auction is created within this network, a random node is chosen to be the "TTP", using a hash of the auction data. If the node is random and the network is large, there is only a small chance of choosing a malicious attacker. Then, the randomly chosen TTP can be found by "traversing" the DHT. Probability of collusion can be rapidly reduced by choosing multiple random nodes to be TTPs, as above. Choosing multiple nodes as TTPs, using separate hash functions to produce the random nodes, also avoids the possibility of malicious sellers manipulating their auction data to hash to their own node, as the probability of each hash selecting a colluding node rapidly lowers.

9. Conclusions

We show that our proposed framework can be used to run distributed private value auctions. Cryptographically secure multiparty computation (where each bidder is a party to the protocol) ensures that only the highest bid and the identity of the highest bidder is revealed to the seller. Publicly verifiable proofs of each step of the computation during the auction are published. This allows third parties (apart from the bidders and the seller) to verify the result of the auction. More importantly, both our proposed framework and the library implementation prevent all attacks on such distributed private value auctions described previously [8].

The techniques presented in this paper can be used to design and run large auctions (such as spectrum allocation, auction of natural resources, etc.) that are subject to oversight by third-party verifiers (i.e., the public), while ensuring integrity and secrecy of all losing bids, and thereby protecting private interests of the bidding parties:

1. Every bidder provides the address of her Bitcoin wallet while signing up for the auction. The seller also provides the address of her wallet while joining the auction. When the auction ends, a simple Bitcoin script can ensure that the funds are transferred from the winner to the seller. Such a system would benefit greatly from unifying the private keys used by Bitcoin (necessary to spend funds) and the ones used by the auction protocol.
2. An implementation of the proposed library that can be ported to mobile devices can be used to build voting mechanisms and group decision (or social choice) protocols that can run as individual applications. Veto voting as described by Brandt [6] is an example of one such group decision protocol.
3. Using a distributed hash table (or ledger) may possibly eliminate the need for a non-random trusted third party to distribute certificates to bidders prior to the auction.

Acknowledgments: We would like to acknowledge Tim Roughgarden (Stanford University) for advising us, and would like to also acknowledge Felix Brandt for his many years of work designing the theory behind our implementation.

Author Contributions: Each author contributed significantly to the design of the distributed application, the implementation, the analysis of the implementation's advantages and disadvantages, and to writing and editing the paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Franklin, M.K.; Reiter, M.K. The design and implementation of a secure auction service. *IEEE Trans. Softw. Eng.* **1996**, *22*, 302–312.
2. Lipmaa, H.; Asokan, N.; Niemi, V. *Secure Vickrey Auctions without Threshold Trust*; Springer: Berlin/Heidelberg, Germany, 2003; pp. 87–101.
3. Naor, M.; Pinkas, B.; Sumner, R. Privacy Preserving Auctions and Mechanism Design. In Proceedings of the 1st ACM Conference on Electronic Commerce (EC '99), Denver, CO, USA, 3–5 November 1999; ACM: New York, NY, USA, 1999; pp. 129–139.

4. Harkavy, M.; Tygar, J.D.; Kikuchi, H. Electronic Auctions with Private Bids. In Proceedings of the USENIX Workshop on Electronic Commerce, Boston, MA, USA, 31 August–3 September 1998.
5. Bogetoft, P.; Damgård, I.; Jakobsen, T.P.; Nielsen, K.; Pagter, J.; Toft, T. A practical implementation of secure auctions based on multiparty integer computation. In *Financial Cryptography*; Springer: Berlin/Heidelberg, Germany, 2006; Volume 4107, pp. 142–147.
6. Brandt, F. *Efficient Cryptographic Protocol Design Based on Distributed El Gamal Encryption*; Springer: Berlin/Heidelberg, Germany, 2006; pp. 32–47.
7. Brandt, F. How to obtain full privacy in auctions. *Int. J. Inf. Secur.* **2006**, *5*, 201–216, doi:10.1007/s10207-006-0001-y.
8. Dreier, J.; Dumas, J.; Lafourcade, P. Brandt's fully private auction protocol revisited. *J. Comput. Secur.* **2015**, *23*, 587–610, doi:10.3233/JCS-150535.
9. Fiat, A.; Shamir, A. *How to Prove Yourself: Practical Solutions to Identification and Signature Problems*; Springer: Berlin/Heidelberg, Germany, 1987; pp. 186–194.
10. Abdalla, M.; An, J.H.; Bellare, M.; Namprempre, C. *From Identification to Signatures via the Fiat–Shamir Transform: Minimizing Assumptions for Security and Forward-Security*; Springer: Berlin/Heidelberg, Germany, 2002; pp. 418–433.
11. ElGamal, T. *A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms*; Springer: Berlin/Heidelberg, Germany, 1985; pp. 10–18.
12. Tsiounis, Y.; Yung, M. *On the Security of ElGamal Based Encryption*; Springer: Berlin/Heidelberg, Germany, 1998; pp. 117–134.
13. Schnorr, C.P. *Efficient Identification and Signatures for Smart Cards*; Springer: Berlin/Heidelberg, Germany, 1990; pp. 688–689.
14. Cramer, R.; Gennaro, R.; Schoenmakers, B. *A Secure and Optimally Efficient Multi-Authority Election Scheme*; Springer: Berlin/Heidelberg, Germany, 1997; pp. 103–118.
15. Groth, J.; Lu, S. *Verifiable Shuffle of Large Size Ciphertexts*; Springer: Berlin/Heidelberg, Germany, 2007; pp. 377–392.
16. Yao, A.C. Protocols for Secure Computations. In Proceedings of the 23rd Annual Symposium on Foundations of Computer Science (SFCS '82), Chicago, IL, USA, 3–5 November 1982; IEEE Computer Society: Washington, DC, USA, 1982; pp. 160–164.
17. Kulshrestha, A.; Rampuria, A.; Denton, M.; Sreenivas, A. Auctions. 2016. Available online: https://github.com/ashwinsr/auctions (accessed on 10 December 2017).