# VTBC: Privatizing the Volume and Timing of Transactions for Blockchain Applications

Trevor Miller
*Virginia Tech*
trevormil@vt.edu

Bobby Alvarez
*Virginia Tech*
alvarezbobby9@vt.edu

Thang Hoang
*Virginia Tech*
thanghoang@vt.edu

***Abstract*—Existing privacy-preserving blockchain solutions have shown how to maintain the anonymity and confidentiality of the contents of blockchain transactions. However, due to blockchains needing to be stored and updated in a decentralized manner, metadata like the volume of transactions and the timestamp of each transaction can always be publicly observed, even with state-of-the-art solutions. Blockchain applications, especially ones with time-sensitive or volume-sensitive outcomes, may require this volume and timing information to be privatized. One example is not leaking the lateness of students' exam submissions because this could violate student privacy laws.**

**In this paper, we propose VTBC, a blockchain system to privatize such volume and timing information for multi-party privacy-preserving blockchain applications through decoy blockchain transactions which a) do not contribute at all to the execution of the application and b) are indistinguishable from real (non-decoy) transactions. Even though the volume and timing metadata of all transactions must be public, volume and timing information for an application can be indirectly privatized (even after the application has been finalized) by carefully deciding when and how many decoy transactions are added to the blockchain. We demonstrate how these decoy transactions can be created without sacrificing the application's integrity, functionality, or verifiability, without making changes to the underlying blockchain's architecture, and always using the blockchain as the trusted timekeeper. We implemented our approach via a Dutch auction that supports decoy bid transactions and evaluated its performance on a private Ethereum blockchain network.**

***Index Terms*—blockchain, privacy, zero-knowledge, timestamp**

## I. Introduction

Many real-world sectors such as finance, education, and governance have been touted to be revolutionized by blockchains [1]. This is because blockchains offer these applications a unique set of characteristics that has never been seen before in the digital world. Specifically, blockchains are public, decentralized, distributed, and append-only ledgers of transactions whose current state is continuously agreed upon through the consensus of network participants and not by any centralized party. So for example, the finance sector can transition to using cryptocurrencies like Bitcoin which use blockchains for an immutable, public, decentralized, tamper-proof ledger of digital payment transactions [2]. This offers many advantages over existing currencies such as not being controlled by a single, centralized entity [2].

Despite blockchains having so much potential though, a major obstacle to their adoption is they are prone to leak privacy due to needing to store all their contents in a public, distributed manner. For example, although Bitcoin users transact without revealing identity (using pseudonyms), all payment information is publicly viewable such as every transaction's payment amount, every user's balance, and every user's transaction history [2]. To address such privacy issues, privacy-preserving blockchains have been proposed to achieve confidentiality of information such as balances and payment amounts stored on the decentralized ledger. Existing privacy-preserving blockchains have successfully shown how to achieve full anonymity and confidentiality of a transaction's contents while maintaining the integrity and verifiability of the system [3] [4] [5] [6]. For example, the ZCash blockchain uses these techniques to support fully anonymous, private cryptocurrency payments [4].

However, all privacy-preserving blockchains still leak metadata, such as the timing and volume of every transaction. This is because even though the contents of a transaction can be kept confidential and anonymous as explained, each transaction still needs to be posted to the blockchain. And because the blockchain is stored and updated in a distributed manner, anyone can observe when every transaction is posted. In many applications, this volume and timing metadata being leaked can bring about privacy issues and affect the fairness of outcomes, especially when combined with auxiliary information. This is often the case when an application's functionality is time-sensitive.

One such example is students' grades. Blockchain-based grading systems have been proposed that use blockchains to store students' exam submissions, verifiably and trustlessly grade them, and store the outputted grade [7]. In an educational context, a student's privacy over their grades and submissions is important and often enforced by laws and regulations. So in practice, blockchain based grading systems must be able to achieve privacy of students' grades. Let's assume an exam has a typical grading policy that penalizes late submissions and resubmissions. If this grading policy is implemented on a blockchain using existing privacy-preserving techniques, each student could submit their exam verifiably while achieving anonymity and maintaining the confidentiality of their submission(s). However, the public metadata of these submission transactions will leak the timestamps of all submissions and the total amount of submissions. This can then be used to infer the class' performance which is information that is often

wanted to be kept confidential and could be in violation of student privacy laws. To demonstrate, assume a class of $Y$ students, and exams are penalized 25% for being late and 25% for each time the exam is resubmitted (max 3). Students could observe, for example, that there were $2Y$ submissions that were all late which would reveal a maximum class average of 50%.

Another example is a Dutch auction. Dutch auctions are a way to sell one or more items by initially listing them at a high asking price and incrementally lowering the price over time until they have all been sold [8]. The price decrements are typically done at predefined time periods (e.g. the items cost \$10 on day one, \$5 on day two, and so on). In other words, buyers will pay a publicly known price for items dependent on their time of purchase. Because of the properties of such an auction, if the timing and volume of sales are public (as will be the case even when using privacy-preserving blockchains), this will leak certain information which the seller may want to keep confidential. Firstly, the seller's total revenue can be inferred using the public prices in combination with the volume and timing of sales. Alternatively, the current volume of sales can be exploited by buyers to get a lower price. For example, if a bidder is willing to buy an item from a large collection but sees there have been no purchases yet, there is a low probability it will get sold out soon which may tempt them to wait until the price decreases.

Thus, we ask the following question: Given that the timing and volume of transactions for applications on blockchains leaks significant information, can we design a solution for such applications which does not leak volume and timing information while not sacrificing the integrity or functionality of the application?

### A. Our Contributions

We answer this question by proposing VTBC, a new system for implementing multi-party privacy-preserving blockchain applications. In these applications, multiple parties want to execute some joint computation (i.e. the application's logic) using secret inputs in a manner which does not leak their inputs to others. In addition to maintaining the privacy of parties' inputs for these applications, our approach is able to maintain volume and timing privacy such that the volume and timestamps of the application's blockchain transactions will leak no private information, even after the application has been finalized. This makes our approach ideal for applications with time-sensitive or volume-sensitive outcomes (e.g. a student exam or Dutch auction). We achieve this in a manner where anyone can verify the application's execution to be correct, fair, and honest. To our knowledge, we are the first paper to focus on maintaining volume and timing privacy.

## II. Preliminaries

### A. Notation

Let $||$ represent concatenation. We denote $[N] = \{1, ..., N\}$. Let a Probabilistic Polynomial Time (PPT) adversary be an adversary who is computationally bounded and can only execute PPT computations.

We denote $\mathcal{E}$ = (Gen, Enc, Dec) as symmetric key encryption where $k \leftarrow \mathcal{E}.\mathsf{Gen}(1^\lambda)$ generates a symmetric key $k$ with a security parameter $\lambda$, $c \leftarrow \mathcal{E}.\mathsf{Enc}(k, m)$ encrypts plaintext $m$ with the key $k$, and $m \leftarrow \mathcal{E}.\mathsf{Dec}(k, c)$ decrypts the ciphertext $c$ with the key $k$.

We denote $\mathcal{E}'$ = (Gen, Enc, Dec) as asymmetric key encryption where $(\mathsf{pk_{enc}}, \mathsf{sk_{enc}}) \leftarrow \mathcal{E}'.\mathsf{Gen}(1^\lambda)$ generates a a unique key pair $(\mathsf{pk_{enc}}, \mathsf{sk_{enc}})$ with a security parameter $\lambda$, $c \leftarrow \mathcal{E}'.\mathsf{Enc}(\mathsf{pk_{enc}}, m)$ encrypts plaintext $m$ with the public key $\mathsf{pk_{enc}}$, and $m \leftarrow \mathcal{E}'.\mathsf{Dec}(\mathsf{sk_{enc}}, c)$ decrypts the ciphertext $c$ with the secret key $\mathsf{sk_{enc}}$.

Let $\Sigma$ = (Gen, Sign, Verify) represent cryptographic digital signatures where $(\mathsf{pk_{sig}}, \mathsf{sk_{sig}}) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$ generates a key pair, $\sigma \leftarrow \Sigma.\mathsf{Sign}(\mathsf{sk_{sig}}, m)$ produces a signature $\sigma$ for plaintext $m$ using $\mathsf{sk_{sig}}$, and $\{0, 1\} \leftarrow \Sigma.\mathsf{Verify}(\mathsf{pk_{sig}}, m, \sigma)$ verifies whether the signature $\sigma$ of $m$ is valid (1) for $\mathsf{pk_{sig}}$ or not (0).

We denote Txn = (Create, Parse, Timestamp) to represent select functionality of blockchain transactions. $(\mathsf{txn}, \sigma) \leftarrow \mathsf{Txn.Create}(\mathsf{pk_{sig}}, \mathsf{sk_{sig}}, m)$ constructs a well-formed blockchain transaction txn associated with $\mathsf{pk_{sig}}$ where the transaction's contents consist of the message $m$ and $\sigma$ is the transaction's signature obtained from $\sigma \leftarrow \Sigma.\mathsf{Sign}(\mathsf{sk_{sig}}, m)$. $(\mathsf{pk_{sig}}, m) \leftarrow \mathsf{Txn.Parse}(\mathsf{txn})$ parses a transaction txn to get its respective $\mathsf{pk_{sig}}$ and $m$. Informally, $\mathsf{pk_{sig}}$ can be thought of as a public blockchain pseudonym. $(\sigma, \mathsf{txn}, y) \leftarrow \mathsf{Txn.Timestamp}(\sigma, \mathsf{txn})$ assigns the current timestamp $y$ to the inputted transaction $(\sigma, \mathsf{txn})$. The tuple $(\sigma, \mathsf{txn}, y)$ represents a complete blockchain transaction.

### B. Zero-Knowledge Proofs and zkSNARKs

In this paper, we use zero-knowledge Succinct Non-interactive ARguments of Knowledge (zkSNARKs). zkSNARKs enable a prover $\mathcal{P}$ to generate a succinct proof $\pi$ which can convince a verifier $\mathcal{V}$ that they know some NP statement is true without revealing anything other than the veracity of the statement [9]. In other words, for an NP relation $\mathcal{R}$, $\mathcal{P}$ can convince $\mathcal{V}$ that it knows a witness $\mathbf{w}$ for some input in an NP language $\mathbf{s} \in \mathcal{L}$ such that $\mathbf{s}, \mathbf{w} \in \mathcal{R}$ and $\mathcal{R}(\mathbf{s}, \mathbf{w}) = 1$. zkSNARKs are widely used within the blockchain industry today [9] [10].

zkSNARKs satisfy the following properties: a) completeness — $\mathcal{P}$ can convince $\mathcal{V}$ of any true statement, b) soundness — a malicious $\mathcal{P}$ can not convince $\mathcal{V}$ of a false statement, and c) zero-knowledge — nothing else is revealed to $\mathcal{V}$ besides that the statement is true.

Formally, the functionality of zkSNARKs consists of three algorithms as zkp = (Setup, Prove, Verify) [11]:

- $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \mathcal{R})$: Given a security parameter $\lambda$ and an NP relation $\mathcal{R}$, the public parameters pp are generated.
- $\pi \leftarrow \mathsf{Prove}(\mathsf{pp}, \mathbf{s}, \mathbf{w})$: Given pp, a public statement $\mathbf{s}$, and a private witness $\mathbf{w}$ where $\mathbf{s}, \mathbf{w} \in \mathcal{R}$ and $\mathcal{R}(\mathbf{s}, \mathbf{w}) = 1$, a proof $\pi$ is generated.

- $\{0,1\} \leftarrow \mathsf{Verify}(\mathsf{pp}, \mathbf{s}, \pi)$: Given $\mathsf{pp}$, the proof $\pi$, and the public statement $\mathbf{s}$ (but not $\mathbf{w}$), return if the proof is valid (1) or invalid (0).

## III. Problem Overview

As explained in the introduction, the volume and timing metadata of blockchain transactions is always publicly leaked, even with state-of-the-art privacy-preserving solutions. This metadata being leaked is problematic for time-based or volume-based blockchain applications (i.e. applications whose functionality directly depends on the volume and timing of its transactions). This is because such applications often have privacy requirements which are only satisfied if volume and timing metadata can be kept private (e.g. student exam or Dutch auction).

Currently, volume-based and time-based applications cannot currently be implemented on blockchains in a privacy-preserving manner due to this metadata leakage. All blockchain applications are forced to accept that all volume and timing metadata will be public and design their application with this in mind. If this is not acceptable for an application, they are not able to use a blockchain altogether. Thus, there is a clear need to demonstrate a solution to privatize this metadata for blockchain applications.

In this section, we will explore how such metadata being public can lead to severe consequences for blockchain applications such as violating privacy laws, affecting the fairness of an application's outcome, or leaking other auxiliary information.

### A. Exploiting Pending Transactions

In multi-party applications, this metadata can potentially be used to manipulate the application and gain a more favorable outcome for themselves. Applications are especially prone to be manipulated if this information can be learned while the transactions are still pending (i.e. consensus has not been reached yet).

To demonstrate, assume a user is purchasing a large number of digital collectibles on a blockchain (e.g. Non-Fungible Tokens or NFTs), and all collectibles are listed on a public marketplace with public prices by their respective owners.

If bots can observe a high volume of pending purchase transactions at one time, they can attempt to exploit this by front-running these purchases. This would result in failed purchases or less optimal prices for the original buyer, and the bots can realize profit at the expense of the original buyer. In fact, there is a whole industry of these bots on blockchains currently called maximal extractable value (MEV) bots. These bots observe the public volume and timing metadata of an applications' pending transactions and add them to the blockchain in an order that is most favorable or profitable for them. It is estimated that these MEV bots have maliciously profited $674 million from users since 2020 on the Ethereum blockchain [12].

Although certain information in this example can be privatized if using privacy-preserving blockchains (e.g. what items were purchased at what prices), the volume and timing of all sales for the marketplace will be still publicly available. With auxiliary information, MEV exploitation is still possible. For example, if an item is released onto the marketplace at 7PM and a bot observes a high number of pending purchase transactions starting at that time, it can be concluded with high certainty that the transactions correspond to the newly released item.

### B. Analyzing Confirmed Transactions

Even if transactions can get past the pending queue unobserved, multi-party applications are still prone to be manipulated through observing timing and volume metadata. For example, in existing sealed-bid privacy-preserving auction implementations on blockchains (i.e. standard auctions which maintain confidentiality of bid amounts), even if bid transactions can be made fully anonymously and maintain the privacy of the bid amount, each bid must still be submitted through a blockchain transaction [3]. Bidders can then analyze the timing and volume of other bid transactions in order to learn information, gain competitive advantages, and exploit the application. For example, if a bidder can observe that there have been no previous bids, they know that a small $1 bid could win. This can be seen in Figure 1.

A seller may also want to keep certain information private such as if demand is low because this could damage their reputation. Another reason a seller may want to privatize the volume and timing of bids is because it has been shown that auctions that hide the volume of bids on average attract more bidders due to greater competition uncertainty [13] [14].

## IV. Models

### A. System Model

Our system consists of $U$ participating users and a manager. We assume that the manager is not a participating user. Each user will have a secret unique user identifier $\mathsf{uid}$ only known to them. We will refer to specific users using $\mathsf{uid}$. Note these identifiers are different from public blockchain pseudonyms.

In our system, each user and the manager communicate with each other through blockchain transactions in order to facilitate a multi-party application with a time-sensitive target function $F$ as

$$\{(o_i, \mathsf{uid}'_i)\}_{i=1}^{Q} \leftarrow F(\{(\mathsf{uid}_j, x_j, y_j, z_j)\}_{j=1}^{N})$$

where $N$ inputs are submitted to the target function $F$ by the $U$ users and each $o_i$ is a secret output value for the application specific to user $\mathsf{uid}'_i$. Each input ($\mathsf{uid}_j$, $x_j$, $y_j$, $z_j$) is a tuple where $\mathsf{uid}_j$ is the $\mathsf{uid}$ of the user submitting the input, $x_j$ is the user's secret input value, $y_j$ is the timestamp of the input assigned by the blockchain, and $z_j \in \{0,1\}$ is the secret bit value denoting if the input is a decoy ($z_j = 0$) or real ($z_j = 1$), respectively (see Figure 2). We say the function $F$ is time-sensitive because the output depends on the time ($y_j$) when users submit their inputs.

Our system consists of three stages $\mathsf{Initialization}$, $\mathsf{Submission}$, and $\mathsf{Finalization}$ spanning across the times $T_0, T_1, T_2$, and $T_3$ as follows.

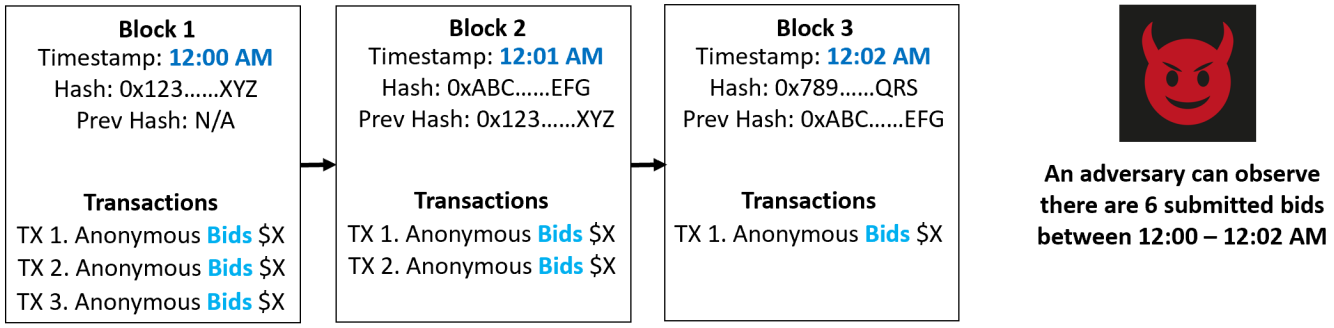


Fig. 1. Example sealed-bid auction implementation on a blockchain that demonstrates how the volume and timing metadata of all bids is leaked, even with full anonymity and confidential bid amounts.

- Initialization ($T_0$ to $T_1$): First, the application is initialized by defining the logic of function $F$, along with other setup to initialize system parameters.
- Submission ($T_1$ to $T_2$): During this stage, each user uid will provide one or more inputs to the common function $F$. To submit an input, they will send ($\text{uid}_j$, $x_j$, $z_j$) to the manager via a blockchain transaction. The blockchain then assigns a public timestamp $y_j$ to this transaction. This creates the tuple ($\text{uid}_j$, $x_j$, $y_j$, $z_j$).
- Finalization ($T_2$ to $T_3$): In this stage, the manager privately executes the function logic $F$ on all the submitted ($\text{uid}_j$, $x_j$, $y_j$, $z_j$) inputs and sends each $o_i$ back to the corresponding user $\text{uid}'_i$ via a blockchain transaction.

### B. Threat Model

In our system, the blockchain is trusted. Specifically, we trust that it will provide a correct timestamp $y_j$ for the users' transactions, ensure data availability, and correctly verify if transactions are well-formed.

Similar to [3], we minimally trust the manager. Specifically, we trust the manager to maintain the privacy of all users' secret input values ($\text{uid}_j$, $x_j$, $z_j$) and secret output values ($o_i, \text{uid}'_i$). We also trust them to privately execute the logic of the target function $F$ and send each secret $o_i$ back to the corresponding user $\text{uid}'_i$. However, we do not trust they will do so correctly. We assume that they may attempt to deviate from the protocol (i.e. excluding/editing/adding inputs to $F$, executing a different computation $F'$, or manipulating $o_i$ values). We assume each user will maintain the privacy of their own secret ($\text{uid}_j$, $x_j$, $z_j$) and ($o_i, \text{uid}'_i$) values. However, we do not trust the user with correctly revealing ($\text{uid}_j$, $x_j$, $z_j$) to the manager.

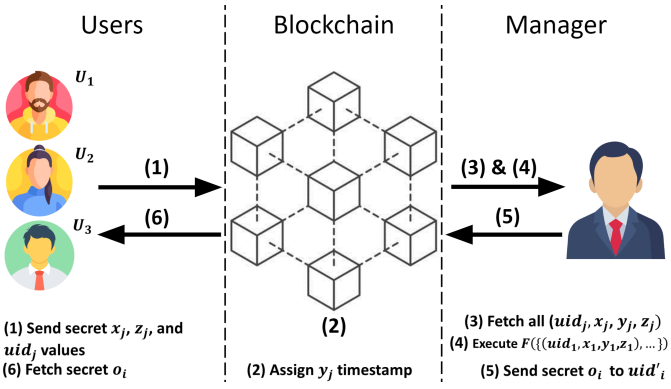


Fig. 2. System model figure that demonstrates the interactions between the participating users, the manager, and the blockchain.

We assume all blockchain nodes can be the adversaries that attempt to learn about the other users' secret input values ($\text{uid}_j$, $x_j$, $z_j$) and secret output values ($o_i, \text{uid}'_i$) using the target function $F$ and all blockchain transactions.

### C. Security Guarantees

To enable security against the aforementioned adversaries, we aim to achieve three security properties: a) input value privacy, b) correctness of execution, and c) volume and timing privacy. The first two are inherited from [3], while volume and timing privacy is our new security feature.

*Input Value Privacy*: Input value privacy is achieved if a PPT adversary can not deduce the values of any secret input ($\text{uid}_j$, $x_j$, $z_j$) or output ($o_i, \text{uid}'_i$) unavailable to them.

*Correctness of Execution*: Even though the users and the manager may attempt to act maliciously and selfishly to maximize their own outcomes, the execution of the application is guaranteed to be correct, fair, and honest.

*Volume and Timing Privacy*: Volume and timing privacy is achieved if a PPT adversary can not learn the volume and timestamps of the real user inputs as well as their impact on the outcome of the target function $F$.

## V. Proposed Method

### A. Overview

In this section, we propose our method to privatize the volume and timing of transactions for blockchain applications in a manner that achieves all of our previously outlined security guarantees. We use a base approach from [3] which utilizes a minimally trusted manager, verifiable zkSNARKs, encryptions, and uniquely generated pseudonyms for each user input submission transaction. We then extend this approach to privatize the volume and timestamps of an application's real

inputs ($z_j = 1$) through the use of decoy inputs ($z_j = 0$). Decoy inputs never affect the application's outcome whereas real ones can. The only purpose of decoy inputs is to add another blockchain transaction to create additional noise. To an adversary, decoy inputs will be kept indistinguishable from real inputs a) by keeping each $z_j$ value encrypted and b) by the design of the target function $F$.

The intuition behind our approach is that if sufficient noise is added through indistinguishable decoy inputs, a PPT adversary will not be able to determine if any user input submission transaction corresponds to a real input or a decoy input, given that they have access to $F$, the $N$ input submission transactions, and the manager's finalization transaction. Thus, even though the volume and timestamps of all transactions can be publicly observed on the blockchain, the volume and timestamps of the application's real inputs can not be learned.

The challenge we face is that even if each $z_j$ can be kept private using the manager, the application-specific design of $F$ in combination with the volume and timing distribution of the $N$ submission transactions oftentimes will leak information to an adversary. For example, let's say we have a deadline-based application (e.g. a student exam) where inputs are deemed on-time or late according to a public deadline during Submission (i.e. deadline is some time between $T_1$ and $T_2$). If there is only one input submitted before the deadline, an adversary can observe the blockchain to learn this (even if $z_j$ is kept private), and this will leak that $\geq U - 1$ users submitted late. Another example is an adversary could conclude that submissions posted close to the deadline are more likely to be real than decoy due to procrastination.

### B. The Application's Target Function, $F$

*1) Designing $F$:* While the target function $F$ and determining the outcome of an application is application-specific, we enforce $F$ to follow a specific structure that protects against an adversary analyzing the volume and timing distribution of the $N$ submission transactions.

Let there be $K$ timestamps $\{ts_1, ts_2, \ldots ts_K\}$ where $T_1 \leq ts_1 \leq \ldots \leq ts_K \leq T_2$ and let $\Delta$ be a short length of time. Let $g$ represent the portion of logic within $F$ which is application-specific and determines the application's outcome using only the real and valid inputs as

$$\{(o_i, \mathsf{uid}'_i)\}_{i=1}^{Q} \leftarrow g(\{(\mathsf{uid}_d, x_d, y_d, z_d)\}_{d=1}^{H})$$

where $H \leq N$. Within $F$, we first filter the original inputs $\{(\mathsf{uid}_j, x_j, y_j, z_j)\}_{j=1}^{N}$ down to $H$ inputs by a) removing decoy inputs and b) removing inputs from users who do not submit at all $K$ time periods ($ts \pm \Delta$). Then, the real and valid inputs (i.e. the filtered $H$ inputs) are inputted to $g$ to compute the outcome $\{(o_i, \mathsf{uid}'_i)\}_{i=1}^{Q}$. Formally, the logic of $\{(o_i, \mathsf{uid}'_i)\}_{i=1}^{Q} \leftarrow F(\{(\mathsf{uid}_j, x_j, y_j, z_j)\}_{j=1}^{N})$ is structured as:

- $\mathcal{I} \leftarrow \{(\mathsf{uid}_j, x_j, y_j, z_j)\}_{j=1}^{N}$
- $\mathcal{U} \leftarrow \{\emptyset\}$
- For each $j \in [N]$, if $\mathsf{uid}_j \notin \mathcal{U}$ and for each $a \in [K]$, $\exists$ $(\mathsf{uid}_b, x_b, y_b, z_b) \in \mathcal{I}$ such that $b \in [N]$, $\mathsf{uid}_b = \mathsf{uid}_j$, and $ts_a - \Delta \leq y_b \leq ts_a + \Delta$: $\mathcal{U} \leftarrow \mathcal{U} \cup \{(\mathsf{uid}_j)\}$

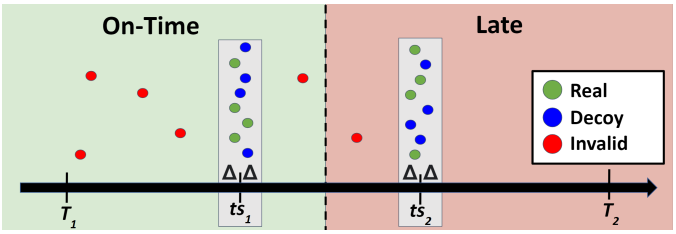


Fig. 3. This figure demonstrates our approach of an example application with a public deadline where users submit at each of $K$=2 time periods. Inputs submitted after the deadline are deemed late. An adversary will try to learn the lateness of the real inputs, but they will not be able to because real (green) and decoy (blue) inputs are indistinguishable to them. Invalid (red) inputs are publicly rejected by the blockchain.

- $\mathcal{I}' \leftarrow \{\emptyset\}$
- For each $j \in [N]$, if $z_j = 1$ and $\mathsf{uid}_j \in \mathcal{U}$: $\mathcal{I}' \leftarrow \mathcal{I}' \cup \{(\mathsf{uid}_j, x_j, y_j, z_j)\}$
- $\{(o_i, \mathsf{uid}'_i)\}_{i=1}^{Q} \leftarrow g(\mathcal{I}')$

The inputs that are not submitted during any $ts \pm \Delta$ are never inputted to $F$ because their corresponding transactions will be publicly rejected by the blockchain (see Section V-C). Through the above logic, only real inputs from users who submit during all $K$ time periods can affect the application's outcome $\{(o_i, \mathsf{uid}'_i)\}_{i=1}^{Q}$.

*2) Length of $\Delta$:* $\Delta$ should be as short as possible but also long enough that users will not fall victim to denial-of-service attacks or get unlucky where their transaction is not confirmed in time by the asynchronous blockchain.

*3) Time-Based and Volume-Based Outcomes:* For applications with time-sensitive or volume-sensitive outcomes (e.g. a student exam or a Dutch auction), the $K$ time periods should span all possible volume and timing combinations that could potentially affect any $o_i$. For example, in a deadline-based application (as discussed in Section V-A), the $o_i$ values are dependent on if the $y_j$ input timestamps are either before or after some public deadline, so in this case, a minimum of $K$=2 time periods are required: one before and one after the deadline (see Figure 3).

### C. Detailed Algorithm

In this section, we present our algorithm in detail. Let $\mathcal{R}$ be the relation for zkp where users generate a proof $\pi$ that they encrypted their secret ($\mathsf{uid}_j$, $x_j$, $z_j$) values to the manager correctly. Let $\mathcal{R}'$ be the relation for zkp where the manager generates a proof $\pi'$ that they finalized the application correctly. This consists of a) proving knowledge of all users' secret ($\mathsf{uid}_j$, $x_j$, $z_j$) values, b) proving $F$ was executed correctly, and c) proving the secret $o_i$ output values were encrypted and sent correctly to each user $\mathsf{uid}'_i$ via the blockchain. We utilize both symmetric $\mathcal{E}$ and asymmetric $\mathcal{E}'$ encryptions during our algorithm so that generating $\pi'$ can be done more efficiently using only $\mathcal{E}$ rather than $\mathcal{E}'$ (see Section VI).

1) Initialization ($T_0$ to $T_1$): First, the manager needs to establish their key pairs, initialize the application's functionality, and initialize all system parameters as:

- $(\mathsf{pk}_{\mathsf{sig_M}}, \mathsf{sk}_{\mathsf{sig_M}}) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$
- $(\mathsf{pk}_{\mathsf{enc_M}}, \mathsf{sk}_{\mathsf{enc_M}}) \leftarrow \mathcal{E}'.\mathsf{Gen}(1^\lambda)$
- $\mathsf{pp} \leftarrow \mathsf{zkp.Setup}(1^\lambda, \mathcal{R})$
- $\mathsf{pp}' \leftarrow \mathsf{zkp.Setup}(1^\lambda, \mathcal{R}')$
- Broadcast $\mathsf{pk}_{\mathsf{sig_M}}$, $\mathsf{pk}_{\mathsf{enc_M}}$, $F$, and any other application-specific logic on the blockchain (including the parameters of pp and pp′ which the blockchain will need during Submission and Finalization to execute zkp.Verify).
- Broadcast pp to all users. The users will locally store pp.

Each user establishes a unique symmetric encryption key:

- $k \leftarrow \mathcal{E}.\mathsf{Gen}(1^\lambda)$

2) Submission ($T_1$ to $T_2$): For a user to submit an input to the target function $F$, they perform the following operations:

- Generate secret values for $x$ and $z$.
- $c \leftarrow \mathcal{E}.\mathsf{Enc}(k, z||x)$
- $c' \leftarrow \mathcal{E}'.\mathsf{Enc}(\mathsf{pk}_{\mathsf{enc_M}}, k)$
- $\mathbf{w} \leftarrow (x,\ z,\ k)$
- $\mathbf{s} \leftarrow (c,\ c',\ \mathsf{pk}_{\mathsf{enc_M}})$
- $\pi \leftarrow \mathsf{zkp.Prove}(\mathsf{pp},\ \mathbf{s},\ \mathbf{w})$
- $(\mathsf{pk}_{\mathsf{sig}}, \mathsf{sk}_{\mathsf{sig}}) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$
- $(\mathsf{txn},\ \sigma) \leftarrow \mathsf{Txn.Create}(\mathsf{pk}_{\mathsf{sig}},\ \mathsf{sk}_{\mathsf{sig}},\ \pi \mid\mid \mathbf{s})$
- Send $(\mathsf{txn},\ \sigma)$ to the blockchain

Upon the blockchain receiving each tuple $(\mathsf{txn}, \sigma)$, the blockchain will execute:

- $(\mathsf{txn}, \sigma, y) \leftarrow \mathsf{Txn.Timestamp}(\mathsf{txn}, \sigma)$
- $(\mathsf{pk}_{\mathsf{sig}},\ \pi \mid\mid \mathbf{s}) \leftarrow \mathsf{Txn.Parse}(\mathsf{txn})$
- $(c,\ c',\ \mathsf{pk}_{\mathsf{enc_M}}) \leftarrow \mathbf{s}$
- If $\{0,1\} \leftarrow \Sigma.\mathsf{Verify}(\mathsf{pk}_{\mathsf{sig}},\ \mathsf{txn},\ \sigma) = 0$ or $\{0,1\} \leftarrow \mathsf{zkp.Verify}(\mathsf{pp},\ \mathbf{s},\ \pi) = 0$ or $\mathsf{pk}_{\mathsf{enc_M}}$ does not match what was defined in Initialization or $y$ does not fall within one of the $K$ time periods outlined in $F$, reject the transaction.
- Else, include $(\mathsf{txn}, \sigma, y)$ on the blockchain.

3) Finalization ($T_2$ to $T_3$): Assume there are $N$ transactions $(\mathsf{txn}_j, \sigma_j, y_j)$ submitted by users during Submission, the manager executes the following for each $j \in [N]$:

- $(\mathsf{pk}_{\mathsf{sig}_j},\ \pi_j \mid\mid \mathbf{s}_j) \leftarrow \mathsf{Txn.Parse}(\mathsf{txn}_j)$
- $(c_j,\ c'_j,\ \mathsf{pk}_{\mathsf{enc_M}}) \leftarrow \mathbf{s}_j$
- $k_j \leftarrow \mathcal{E}'.\mathsf{Dec}(\mathsf{sk}_{\mathsf{enc_M}},\ c'_j)$
- $z_j||x_j \leftarrow \mathcal{E}.\mathsf{Dec}(k_j,\ c_j)$

The manager now uses all the decrypted input values to finalize the application. We will use each user's encryption key $k$ from Initialization as their user identifier uid.

- $\{(o_i, k'_i)\}_{i=1}^{Q} \leftarrow F(\{(k_j, x_j, y_j, z_j)\}_{j=1}^{N})$
- $o'_i \leftarrow \mathcal{E}.\mathsf{Enc}(k'_i,\ k'_i \mid\mid o_i)$ for each $i \in [Q]$
- $\mathbf{w}' \leftarrow (\{(k_j, x_j, z_j)\}_{j=1}^{N}, \{(o_i, k'_i)\}_{i=1}^{Q})$
- $\mathbf{s}' \leftarrow (\{(c_j, y_j)\}_{j=1}^{N}, \{(o'_i)\}_{i=1}^{Q})$
- $\pi' \leftarrow \mathsf{zkp.Prove}(\mathsf{pp}',\ \mathbf{s}',\ \mathbf{w}')$
- $(\mathsf{txn}',\ \sigma') \leftarrow \mathsf{Txn.Create}(\mathsf{pk}_{\mathsf{sig_M}},\ \mathsf{sk}_{\mathsf{sig_M}},\ \pi' \mid\mid \{(o'_i)\}_{i=1}^{Q})$
- Send $(\mathsf{txn}',\ \sigma')$ to the blockchain

Upon receiving $(\mathsf{txn}',\ \sigma')$, the blockchain verifies the application's finalization by executing:

- $(\mathsf{txn}', \sigma', y') \leftarrow \mathsf{Txn.Timestamp}(\mathsf{txn}', \sigma')$
- $(\mathsf{pk}_{\mathsf{sig_M}},\ \pi' \mid\mid \{(o'_i)\}_{i=1}^{Q}) \leftarrow \mathsf{Txn.Parse}(\mathsf{txn}')$
- Fetch $(\mathsf{txn}_j, \sigma_j, y_j)$ from past blocks for $j \in [N]$
- $(\mathsf{pk}_{\mathsf{sig}_j},\ \pi_j \mid\mid \mathbf{s}_j) \leftarrow \mathsf{Txn.Parse}(\mathsf{txn}_j)$ for $j \in [N]$
- $(c_j,\ c'_j,\ \mathsf{pk}_{\mathsf{enc_M}}) \leftarrow \mathbf{s}_j$ for $j \in [N]$
- $\mathbf{s}' \leftarrow (\{(c_j, y_j)\}_{j=1}^{N}, \{(o'_i)\}_{i=1}^{Q})$
- If $\{0,1\} \leftarrow \Sigma.\mathsf{Verify}(\mathsf{pk}_{\mathsf{sig_M}},\ \mathsf{txn}',\ \sigma') = 0$ or $\{0,1\} \leftarrow \mathsf{zkp.Verify}(\mathsf{pp}',\ \mathbf{s}',\ \pi') = 0$ or $\mathsf{pk}_{\mathsf{sig_M}}$ does not match the one defined in Initialization, reject the transaction.
- Else, include $(\mathsf{txn}', \sigma', y')$ on the blockchain.

Each user now has access to their respective output values by:

- Fetch $(\mathsf{txn}', \sigma', y')$ from the blockchain
- $(\mathsf{pk}_{\mathsf{sig_M}},\ \pi' \mid\mid \{(o'_i)\}_{i=1}^{Q}) \leftarrow \mathsf{Txn.Parse}(\mathsf{txn}')$
- $k'_i \mid\mid o_i \leftarrow \mathcal{E}.\mathsf{Dec}(k,\ o'_i)$ for $i \in [\mathrm{Q}]$
  - If $k'_i = k$, this output is for them.

### D. Security Analysis

*1) Input Value Privacy:* To achieve input value privacy, we always keep each user's secret ($\mathsf{uid}_j$, $x_j$, $z_j$) and $(o_i, \mathsf{uid}'_i)$ values encrypted on-chain between them and the manager, who is trusted for privacy, even after the application has been finalized. The manager never leaks such values during Finalization because the execution of $F$ and encrypting each $o_i$ to $\mathsf{uid}'_i$ (i.e. $k'_i$) is done privately but verifiably via $\pi'$. Additionally, users generate unique blockchain pseudonyms ($\mathsf{pk}_{\mathsf{sig}}$) for every blockchain transaction so that no two transactions can be publicly linked.

The above techniques to achieve input value privacy were inherited from [3], but our newly proposed approach of enforcing all $U$ users to submit at $K$ time periods improves upon these techniques. This is because as long as $\Delta$ is sufficiently short, our approach adds sufficient noise such that an adversary can not deduce any secret value using the volume and timing distribution of the $N$ submission transactions (see Figure 3). For example, previously without any time periods, an adversary may have been able to conclude that inputs submitted by users close to a deadline are more likely to be real ($z_j$ = 1) than decoy ($z_j$ = 0) due to procrastination.

*2) Correctness of Execution:* The application's correctness of execution is verified by the trusted blockchain to be correct, fair, and honest through verifying a) $\pi$ from each user which proves all inputs were revealed to the manager correctly and b) $\pi'$ from the manager which proves the finalization of the application was correct. Due to lack of space, our algorithm did not explicitly handle the aborting manager case (i.e. no valid $\pi'$ provided by $T_3$), but our approach can be extended by enforcing a financial penalty if the manager aborts as seen in [3].

Additionally, we use the public $y_j$ timestamps assigned by the blockchain (the trusted timekeeper). This is important to maintain the integrity of our time-sensitive target function $F$.

*3) Volume and Timing Privacy:* We maintain volume and timing privacy through the use of decoy inputs. Decoy inputs are kept indistinguishable from real inputs to an adversary as explained in Section V-D1. Decoy inputs never affect the application's outcome because within $F$, they are filtered out and never get inputted to $g$ which is used to calculate the secret application-specific output $\{(o_i, \mathsf{uid}'_i)\}_{i=1}^{Q}$. $F$ is also executed

privately but verifiably via $\pi'$. Because of these properties, an adversary can not learn the volume and timestamps of the real transactions which impacted the outcome of $F$ (see Figure 3).

For applications with time-sensitive or volume-sensitive outcomes, we set the $K$ time periods to span all possible volume and timing combinations that could affect any $o_i$ (see Section V-B3). During all such time periods, we enforce all $U$ users to submit at least one input, and these inputs can not be distinguished to be decoy or real by an adversary. This adds sufficient noise such that an adversary can not deduce any secret time-sensitive or volume-sensitive output $o_i$ using $F$ and the volume and timing distribution of the $N$ submission transactions (see Figure 3).

*4) User Amount Privacy:* Lastly, we also want to note that if $U$ is not explicitly public, an adversary can not learn the exact value of $U$ due to the indistinguishability of decoy inputs, unique pseudonyms being generated by users, and no maximum on the number of inputs submitted per user per each of the $K$ time periods. For example, in Figure 3, an adversary can observe there is a maximum of eight users, but in practice, there can be anywhere from one to eight users.

This may be beneficial in the case, for example, where a teacher would want to submit decoy inputs and obfuscate on behalf of all students. The responsibility of submitting at each of the K time periods would then be on the teacher instead of the students.

## VI. Implementation and Experiment

### A. Implementation

We implemented our proposed method in Solidity, JavaScript, and Java in about 4000 lines of code. First, we generated our zkSNARK circuit files and circuit input files with xjsnark [15] for the user's submission proof and manager's finalization proof. We term these circuits zkSubmit and zkFinalizeN where $N$ is the total number of user inputs submitted during an application.

For symmetric key encryption, we used AES. For asymmetric key encryption, we used RSA. We used the RSA and AES gadgets provided by xjsnark [15]. The setup (zkp.Setup) and proof generation (zkp.Prove) was done with the libsnark and jsnark libraries [16] [17]. Specifically, we built libsnark with the MULTICORE (multi-threaded option) and ALT_BN_128 flags turned on and ran it with the Groth16 proof system option [18].

Within zkFinalizeN, the manager proves knowledge of each user's inputs using AES encryption as $c \leftarrow \mathcal{E}.\mathsf{Enc}(k, z||x)$. This is more efficient within zkSNARKs compared to RSA decryption $k \leftarrow \mathcal{E}'.\mathsf{Dec}(\mathsf{sk}_{\mathsf{enc_M}}, c')$ and then AES decryption $z||x \leftarrow \mathcal{E}.\mathsf{Dec}(k, c)$ which is what they execute during Finalization to originally get the users' inputs.

For our blockchain, we used a local Ethereum testnet using the HardHat framework and Geth consisting of 10 Ethereum nodes initialized with a blank genesis state and producing blocks every $\sim$12 seconds [19] [20].

We exported each zkSNARK circuit's verification logic to Solidity using EthSnarks which uses a modified version of Groth16 designed for Ethereum [21] [18]. Each circuit's verification logic (zkp.Verify) was deployed as its own Solidity library on our blockchain with one function: verifyProof.

We then deployed two Solidity smart contracts that implemented our proposed method: a generic contract and a Dutch auction contract. For both applications, $N$ was equal to $Q$ since we returned an output corresponding to every input. The generic contract had no application-specific logic and returned an output $o_i$ for each user input where $o_i$ was simply the respective input's $y_j$ timestamp. The Dutch auction contract simulated a Dutch auction environment where a collection of $J$ items were for sale. These items started at some publicly known price, $P$. At the end of each of the $K$ time periods, the price of the items decreased publicly by $P$ / $K$. All bidders submitted one real or decoy bid every time period ($K$). Upon finalization, we returned an output $o_i$ for each bid where $o_i$ was whether the bid was successful or not. A bid was successful if it was not a decoy bid and the items for sale had not sold out yet.

Both contracts had three functions to represent the three stages in our proposed method: Initialize, SubmitInput, and Finalize. Initialize initialized all deadlines, declared who the manager was, and defined the application's logic. SubmitInput verified the user's inputted submission proof with verifyProof from the zkSubmit library and then stored the AES + RSA ciphertexts and the timestamp to storage. Finalize verified the manager's inputted finalization proof with verifyProof from the respective zkFinalizeN library and asserted the AES ciphertexts and timestamps used in the proof exactly matched the ones previously stored.

### B. Hardware

For the user and blockchain, we used a desktop with a 12-core 12th Gen Intel Core i5-12400, 32 GB RAM, and a 1 TB SSD. For the manager, we used a server with a 48-core Intel Xeon Platinum 8360Y CPU @ 2.40 GHz, 128 GB RAM, and a 1.6TB SSD.

### C. Parameters

We tested each application with the total number of user inputs $N$ = 8, 16, 32, 64, and 128. For encryptions, we used a 128-bit AES key (128-bit security) and a 2048-bit RSA modulus (112-bit security). We used the ALT_BN128 elliptic curve for our zkSNARKs (100-bit security) [22] [23].

### D. Metrics

We measured the Ethereum gas used, Ethereum storage used, and end-to-end delay (local proof generation + blockchain confirmation time). For the zkSNARK circuits, we reported the proving/verification key size, proof size, constraints, public inputs, and proving/verification/keygen time. The proving and verification keys are obtained from pp and are used for zkp.Prove and zkp.Verify, respectively. Constraints correspond to the size of the circuit. Public inputs correspond to the size of the statement s.

TABLE I
PROOF, KEY, AND LIBRARY GENERATION OVERHEAD FOR OUR ZKSNARK CIRCUITS.

| | Circuit Size | | zkp.Setup | | | zkp.Prove | | zkp.Verify | Verification Library | |
|---|---|---|---|---|---|---|---|---|---|---|
| Circuit | Constraints | Public Inputs | Time (s) | PKey (MB) | VKey (KB) | Proof (KB) | Time (s) | Time (ms) | Deploy Gas | Bytecode (KB) |
| zkSubmit | 105775 | 69 | 2.30 | 20.58 | 2.91 | 2.34 | 0.84 | 4.8 | 2851586 | 12.648 |
| zkFinalize8 | 226568 | 73 | 2.99 | 33.65 | 3.07 | 2.46 | 1.14 | 4.6 | 2963548 | 13.15 |
| zkFinalize16 | 453136 | 145 | 5.67 | 67.30 | 5.94 | 4.77 | 2.20 | 4.6 | 4994336 | 22.33 |
| zkFinalize32 | 906272 | 289 | 8.08 | 134.60 | 11.68 | 9.375 | 4.33 | 4.6 | 9103364 | 40.88 |
| zkFinalize64 | 1812544 | 577 | 14.35 | 269.21 | 23.17 | 18.59 | 8.62 | 4.7 | 18119193 | 81.67 |
| zkFinalize128 | 3625088 | 1153 | 34.48 | 538.42 | 46.14 | 37.02 | 17.07 | 9.7 | 35333875 | 159.268 |

TABLE II
BLOCKCHAIN OVERHEAD OF THE THREE STAGES FROM OUR PROPOSED METHOD FOR APPLICATIONS WITH $N$ INPUTS ($N$ = REAL + DECOY).

| | Initialization | | Submission | | Finalization | | Total | |
|---|---|---|---|---|---|---|---|---|
| $N$ Inputs | Gas | Bytecode (KB) | Gas | Storage (KB) | Gas | Storage (KB) | Gas | Storage (KB) |
| 8 | 1270118 | 4.28 | 19005140 | 36.384 | 988195 | 2.46 | 21263453 | 43.124 |
| 16 | 1270118 | 4.28 | 37993180 | 72.768 | 1737700 | 4.77 | 41000998 | 81.818 |
| 32 | 1270130 | 4.28 | 75969260 | 145.536 | 3237560 | 9.375 | 80476410 | 159.191 |
| 64 | 1270130 | 4.28 | 151921420 | 291.072 | 6240330 | 18.59 | 159431880 | 313.902 |
| 128 | 1270130 | 4.28 | 303825740 | 582.144 | 12258474 | 37.02 | 317354344 | 623.444 |

### E. Results

*1) Generic Application:* To evaluate, we generated and deployed our verification libraries for each of our circuits (Table I). Then, we simulated the roles of all users and the manager for all three stages from our algorithm (Table II).

*Circuit Setup and Library Deployment:* Table I shows the costs of setting up our circuits and deploying our verification libraries on the blockchain. The zkSubmit circuit's constraints were dominated by the one RSA (∼89000 constraints) and one AES encryption (∼14150 constraints). The zkFinalizeN circuits' constraints were dominated by the $2N$ AES encryptions (i.e. for proving knowledge of the $N$ inputs and for encrypting each of the $N = Q$ outputs). This is why the metrics for these circuits grow linearly as $N$ grows.

For our zkFinalize8 circuit, key generation takes 2.99 seconds, and the generated proving and verification key sizes are 33.65 MB and 3.07 KB, respectively. While the proving key is large, this is stored locally. On the other hand, the verification key is needed to be entirely stored on-chain within our libraries to implement verifyProof. These libraries only needed to store the verification key from pp.

Due to the verification keys being kilobytes in size, our libraries' bytecode sizes are largely dominated by the storage of the verification key. For example, the storage of the verification key alone takes up 96% of our lines of Solidity code for the zkFinalize128 library.

Deployment of our libraries gets more expensive when $N$ increases. For zkFinalize32, zkFinalize64, and zkFinalize128, deployment exceeded predefined Ethereum mainnet network limits (30,000,000 gas limit per block and 24.576 KB bytecode size) [24]. Although, Ethereum does recognize these limits are low for applications like ours using zkSNARKs and are working on solutions such as EIP-196 to allow zkSNARKs on Ethereum to be implemented more efficiently [25].

*Overhead per Stage:* Table II shows the blockchain overhead for each of our three stages from our proposed method. For $N$ = 8, the initialization stage (deploying the contract and calling Initialize) cost 1270118 gas and used 4.28 KB storage. This cost was almost the exact same for all $N$ values because the only small change needed in the code was editing which zkFinalizeN library to call.

Before calling SubmitInput or Finalize, local proof generation is required. For the user's submission proof, this took 0.84 seconds. For the manager's finalization proof, this took from 1-17 seconds depending on $N$. Verification of these proofs only took 4-9 milliseconds.

Each user's execution of SubmitInput cost 2373505 gas and used 4.548 KB of storage. This storage number includes the 2.34 KB proof and 2.208 KB for storing the RSA ciphertext, AES ciphertext, and the timestamp to the contract's storage (which consisted of 69 uint256 integers). The overhead for the submission stage increases linearly with $N$ because applications with more inputs require more submissions.

The manager's execution of Finalize when $N = 8$ cost 988195 gas and used 2.46 KB storage (finalization proof was 2.46 KB). This grows linearly with $N$ due to the increasing proof size.

*2) Dutch Auction:* In this section, we show how our system performed when used by a real-world application: a Dutch auction. We tested Dutch auctions with $K$ = 1, 2, and 4 time periods and up to 128 bids. We set the number of items for sale $J$ = 8.

Figure 4 shows that the total end-to-end delay for each bidder only depends on the number of bids they submit and is independent of how many other bidders there are. When each bidder only submits one bid ($K$ = 1), it took around seven seconds on average (∼0.8 seconds for proof generation + ∼6 seconds for blockchain confirmation). When each bidder submitted two and four bids, the delay increased linearly to

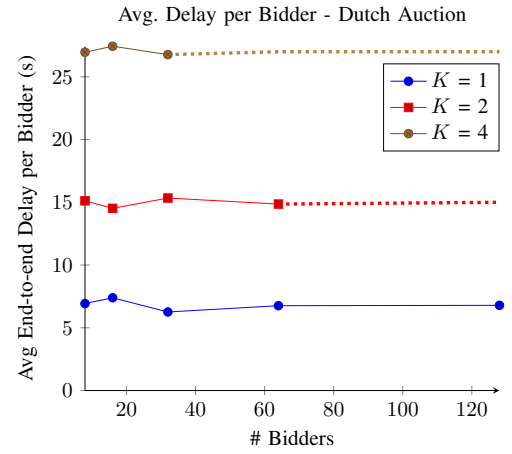


Fig. 4. This figure shows the average end-to-end time delay (proof generation + blockchain confirmation) per bidder in a Dutch auction. We assume each bidder submits a bid (real or decoy) every time period ($K$).

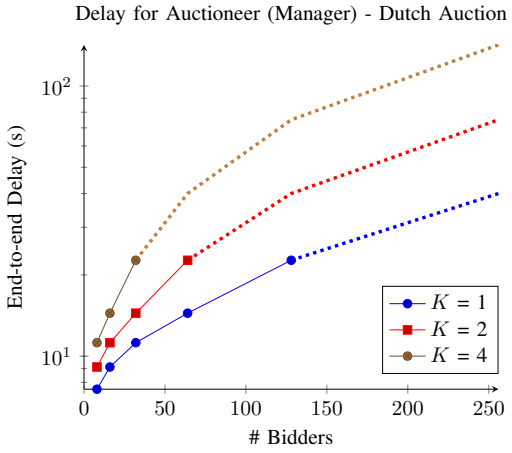


Fig. 5. This figure shows the end-to-end time delay (proof generation + blockchain confirmation) for the auctioneer in a Dutch auction as the number of bidders increases. We assume each bidder submits a bid (real or decoy) every time period ($K$). We extended this figure using projected values to better display our results.

$\sim$14 and $\sim$27 seconds.

Blockchain confirmation time converged to an average of six seconds as we ran the experiment more times (25 runs in total). This is because broadcasting the pending transaction to all nodes only took a few milliseconds, but blocks were confirmed every 12 seconds. So in practice, each confirmation was a random value between 0 and 12 seconds.

Figure 5 shows that the total delay for the auctioneer is directly correlated to the number of bids. We extended Figure 5 using projected values to better display our results. The end-to-end delay is initially dominated by the blockchain confirmation time for small $N$ values. However, the manager's proof generation time and thus end-to-end delay grows linearly with more bids. This is why the end-to-end delay doubles from $K = 1$ to $K = 2$ as well as from $K = 2$ to $K = 4$.

### F. Scalability and Optimizations

We do note that blockchains have scarce time and space resources, and our approach only contributes to this problem through adding additional decoy transactions. For our largest experiment ($N$ = 128), the blockchain only experiences a storage overhead of $<$ 1MB and roughly 10ms per verification. This makes our approach suitable for many private or permissioned blockchain environments. For public blockchains like Ethereum where resources are more scarce, our approach would add significant overhead. We envision that as blockchain scalability, throughput, and the efficiency of zkSNARKs improve over time, the overhead of our approach will become reasonable enough for any blockchain (public or private) to support our solution.

## VII. Related Work

### A. Privacy Preserving Blockchain Applications

We categorized the current solutions for privacy-preserving blockchain applications into three categories: minimally trusted manager, secure multi-party computation (sMPC), and transaction-based. We believe our work is complementary to all three categories such that each of these categories can be extended to support decoy transactions. We showed how to use a minimally trusted manager to support decoy transactions. Some works in the minimally trusted manager category include [3] and [26].

In sMPC, multiple parties want to perform some joint computation without revealing their secret input values to one another. Instead of using a manager, every party performs a portion of the overall computation using their secret inputs in a manner that does not leak them [27] [28]. We envision that sMPC applications can also support computations with decoy inputs, similar to our algorithm.

Lastly, we have the "transaction based" approach which is similar to what is found in public blockchain frameworks, such as Ethereum. Each transaction triggers some on-chain public computation to be executed and can even update application-specific storage in some way. These works attempt to maintain the confidentiality of certain variables, storage, and information during each transaction's execution [29] [5]. We believe this approach will be the most difficult to add support for decoy inputs because there are many blockchain properties that have to be accounted for that could leak whether a transaction was real or decoy, such as which storage slot gets updated or which portion of some code was executed.

### B. Hiding Timestamps

Works like [30] have shown how to create multi-party commitments that can only be revealed after a certain time. While this can be used for multi-party applications like ours, the commitments are required to eventually be revealed in order to finalize the application, which eventually leaks volume and timing information. Our approach can privatize the volume and timing of transactions even after the application has been finalized.

Works like [31] have shown that you can create anonymous tamper-proof timestamps for off-chain digital content such as documents, papers, and videos. However, these approaches can not be done anonymously and privately for on-chain transactions, only for off-chain hashes or other off-chain cryptographic commitments.

### C. Privatizing Which Computation Was Performed

We do note that in a framework where on-chain computations can be verifiably executed without leaking which on-chain logic was executed, there would be no need for a solution such as ours because transactions could not be linked to a specific application. However, to our knowledge, we have not seen an approach that can do this. [32] has attempted to hide which computation was performed, but their approach is not suited to maintain the integrity and verifiability of many blockchain applications.

## VIII. Conclusion

In this paper, we proposed VTBC, a new approach to implementing multi-party privacy-preserving blockchain applications such that the volume and timing metadata of these applications' transactions never leaks any private information. Our solution solved this by having applications be implemented in a way that supports decoy transactions which are indistinguishable from real transactions by adversaries. This enabled time-sensitive or volume-sensitive applications to be implemented on blockchains in a privacy-preserving manner.

## IX. Acknowledgements

This research is partially supported by an unrestricted gift from Robert Bosch, 4-VA, and the Commonwealth Cyber Initiative (CCI), an investment in the advancement of cyber R&D, innovation, and workforce development. For more information about CCI, visit www.cyberinitiative.org.

## References

[1] T. M. Hewa, Y. Hu, M. Liyanage, S. S. Kanhare, and M. Ylianttila, "Survey on blockchain-based smart contracts: Technical aspects and future research," *IEEE Access*, vol. 9, p. 87643–87662, Mar 2021.

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[3] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," *2016 IEEE Symposium on Security and Privacy (SP)*, 2016.

[4] E. Ben Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," *2014 IEEE Symposium on Security and Privacy*, 2014.

[5] S. Steffen, B. Bichsel, M. Gersbach, N. Melchior, P. Tsankov, and M. Vechev, "Zkay," *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[6] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu, "Zexe: Enabling decentralized private computation," *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[7] P. Ocheja, F. J. Agbo, S. S. Oyelere, B. Flanagan, and H. Ogata, "Blockchain in education: A systematic review and practical case studies," *IEEE Access*, vol. 10, pp. 99 525–99 540, 2022.

[8] Z. Shi, C. de Laat, P. Grosso, and Z. Zhao, "Integration of blockchain and auction models: A survey, some applications, and challenges," *IEEE Communications Surveys & Tutorials*, vol. 25, no. 1, pp. 497–537, 2023.

[9] X. Sun, F. R. Yu, P. Zhang, Z. Sun, W. Xie, and X. Peng, "A survey on zero-knowledge proof in blockchain," *IEEE Network*, vol. 35, no. 4, pp. 198–205, 2021.

[10] S. Simunic, D. Bernaca, and K. Lenac, "Verifiable computing applications in blockchain," *IEEE Access*, vol. 9, p. 156729–156745, 2021.

[11] A. Ozdemir and D. Boneh, "Experimenting with collaborative zk-SNARKs: Zero-Knowledge proofs for distributed secrets," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 4291–4308. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/ozdemir

[12] Flashbots, "Mev explore." [Online]. Available: https://explore.flashbots.net/

[13] J. Levin, "Auction theory," Oct 2004. [Online]. Available: https://www.web.stanford.edu/~jdlevin/Econ\%20286/Auctions.pdf

[14] P. Milgrom and R. J. Weber, "The value of information in a sealed-bid auction," *Journal of Mathematical Economics*, vol. 10, no. 1, p. 105–114, 1982.

[15] A. Kosba, C. Papamanthou, and E. Shi, "xjsnark: A framework for efficient verifiable computation," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 944–961.

[16] Scipr-Lab, "Scipr-lab/libsnark: C++ library for zksnarks." [Online]. Available: https://github.com/scipr-lab/libsnark

[17] A. Kosba, "jsnark: A java library for zk-snark circuits." [Online]. Available: https://github.com/akosba/jsnark

[18] J. Groth, "On the size of pairing-based non-interactive arguments," in *Proceedings, Part II, of the 35th Annual International Conference on Advances in Cryptology — EUROCRYPT 2016 - Volume 9666*. Berlin, Heidelberg: Springer-Verlag, 2016, p. 305–326.

[19] NomicFoundation, "Nomicfoundation/hardhat: Hardhat is a development environment to compile, deploy, test, and debug your ethereum software. get solidity stack traces & console.log." [Online]. Available: https://github.com/NomicFoundation/hardhat

[20] ethereum.org. [Online]. Available: https://geth.ethereum.org/

[21] HarryR, "Harryr/ethsnarks: A toolkit for viable zk-snarks on ethereum, web, mobile and desktop." [Online]. Available: https://github.com/HarryR/ethsnarks

[22] T. Kim and R. Barbulescu, "Extended tower number field sieve: A new complexity for the medium prime case," in *Proceedings, Part I, of the 36th Annual International Cryptology Conference on Advances in Cryptology — CRYPTO 2016 - Volume 9814*. Berlin, Heidelberg: Springer-Verlag, 2016, p. 543–571. [Online]. Available: https://doi.org/10.1007/978-3-662-53018-4_20

[23] Zcash, "Understand the concrete security level of the bn_128 curve in libsnark · issue #714 · zcash/zcash." [Online]. Available: https://github.com/zcash/zcash/issues/714

[24] Ethereum, "Eips/eip-170.md at master · ethereum/eips," Sep 2021. [Online]. Available: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-170.md

[25] C. Reitwiessner, "Eip-196: Precompiled contracts for addition and scalar multiplication on the elliptic curve alt_bn128," Feb 2017. [Online]. Available: https://eips.ethereum.org/EIPS/eip-196

[26] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, jun 2019. [Online]. Available: https://doi.org/10.1109%2Feurosp.2019.00023

[27] T. Kerber, A. Kiayias, and M. Kohlweiss, "Kachina – foundations of private smart contracts," in *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, 2021, pp. 1–16.

[28] A. Banerjee, M. Clear, and H. Tewari, "zkhawk: Practical private smart contracts from mpc-based hawk," in *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, 2021, pp. 245–248.

[29] S. Steffen, B. Bichsel, R. Baumgartner, and M. Vechev, "Zeestar: Private smart contracts by homomorphic encryption and zero-knowledge proofs," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 179–197.

[30] Y. Doweck and I. Eyal, "Multi-party timed commitments," 2020. [Online]. Available: https://arxiv.org/abs/2005.04883

[31] X. Liu, "Zero-overhead private timestamping in bitcoin," Apr 2022. [Online]. Available: https://coingeek.com/zero-overhead-private-timestamping-in-bitcoin/

[32] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu, "Zexe: Enabling decentralized private computation," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 947–964.