IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received July 29, 2020, accepted August 6, 2020, date of publication August 24, 2020, date of current version September 3, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3019000

A Permissioned Distributed Ledger for the US Beef Cattle Supply Chain

TANVIR FERDOUSI, (Graduate Student Member, IEEE), DON GRUENBACHER, (Member, IEEE), AND CATERINA M. SCOGLIO, (Senior Member, IEEE)
Department of Electrical and Computer Engineering, Kansas State University, Manhattan, KS 66506, USA

Corresponding author: Tanvir Ferdousi (tanvirf@ksu.edu)

This work was supported in part by the National Science Foundation under Grant CMMI-1744812, and in part by the Global Food Systems Seed Grant Program, Kansas State University.

ABSTRACT Distributed ledgers using blockchain have gained traction in the supply chain industry due to their unique features of immutability and transparency. They have given people the abilities to solve business problems which were impossible using traditional systems. The US beef cattle industry lacks adequate traceability as most of the farm owners consider such data confidential; possibly harming their businesses if exposed. This article attempts to solve this problem by proposing a smart contract-based supply chain framework using a permissioned blockchain network. This system supports anonymity for the users to protect identities and lets every user store their data locally, while ensuring that the changes are recorded in the chain with cryptographic proofs (hashes). The proposed framework also has methods for the users to perform business transactions and transfer animal-related data to new owners as required. In addition to that, smart contracts have been added to conduct anonymous surveys for data aggregation. The technical contribution of this article is in the system design on how users, data, and communications are handled to maintain data ownership and user privacy while ensuring immutability and confidentiality at different levels of data aggregation. This article also contains an evaluation of the system using integration tests where the outcomes meet the expected design requirements. The framework can be applied to the US beef cattle industry as well as other supply chains with minimal modifications.

INDEX TERMS Blockchain, smart contracts, animal tracing, proof of authority, supply chain management, data security, privacy.

I. INTRODUCTION

Cattle industry in the US does not provide the appropriate traceability for the rapid identification of likely infected cattle during infectious outbreaks. The United States Department of Agriculture (USDA) mandates veterinary inspections for inter-state movements [1]. However, intra-state movement data are kept private by the farm owners, which makes it difficult to trace animal movements. There are several projects which are addressing traceability. CattleTrace was established in 2018 to create an infrastructure for an animal disease traceability system in Kansas [2]. Such projects rely on mutual trust of the participants for keeping data secure which can impede widespread adoption. In this article, we propose a technological solution to the issue of security, privacy, and control of private and shared data using a permissioned blockchain. Blockchain technology is designed to be immutable, transparent, and decentralized, which enables secure communications among parties without the need of intermediate or central authorities. Cryptocurrencies [3] were among the first applications of blockchain which expanded into multiple fields due to some useful features of the blockchain architecture. The possibilities are endless with smart contracts that can execute Turing-complete instruction sets [4], [5].

The associate editor coordinating the review of this manuscript and approving it for publication was Patrick Hung.

II. PRELIMINARIES

A. THE US CATTLE FARM SYSTEM

The beef supply chain consists of several components. One prior work [6] divides the beef cattle operations into four stages: ranch, stocker, feedlot, and packer. We add two more components, distributor and retailer in order to complete

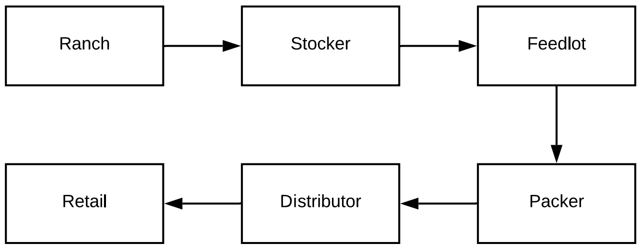


FIGURE 1. A block diagram of the US beef supply chain. The first three (ranch, stocker, and feedlot) segments deal with live animal production. The packer is the manufacturing plant. Once the beef is processed, it reaches the consumers via distributors and retailers.

the chain. The diagram is shown in Figure 1. The cattle stay at different farm types (Ranch, Stocker, and Feedlot) based on their ages and weights. After starting its life in a ranch, a steer or heifer moves into a Stocker when it is 6 to 9 months old and weighs about 400 to 700 pounds. Cattle may move into Feeders for further weight gain. Feeder cattle are aged between 12 to 24 months and weighs about 800 to 1,000 pounds. The time that cattle spend in a feeder is often called the finishing phase. The animals are slaughtered at the Packers where meats are prepared and packed for distribution. We also add a Distributor stage prior to a Retailer as this can be the case in many beef cattle operations. The consumer of the products lies at the end of the supply chain.

B. PRIOR WORK AND MOTIVATION

There have been multiple blockchain based applications for supply chains. Approximately 1.1 billion USD have been invested in blockchain technology [7]. IBM and the Danish shipping company Maersk created a platform named TradeLens and experimented the use of blockchain in order to track shipping containers around the world [8]. As an application in the fishing industry, Provenance launched a project in Indonesia to track tuna using blockchain [9]. Lu *et al.* explored the adaptability of blockchains for product traceability in supply chains and discussed the technology's strengths and limitations [10]. Dudder *et al.* described a model to track timber by using a tamper proof system based on blockchains [11]. Saberi *et al.* discusses different barriers of blockchain technology adoption and identifies the lack of collaboration as one of the inter-organization barriers [12]. A work of Casado-Vara discusses how blockchains can improve traditional linear supply chains into circular economies [13]. Another work developed a provenance knowledge framework and addressed how it can enhance assurances of products quality in the supply chain [14]. Leng *et al.* proposes a dual-chain architecture for use in agricultural supply chains [15]. For traditional supply chains, adoption of this technology would enable traceability and provenance [16]–[19], prevent counterfeits and defects [20], reduce regulatory costs and complexities [21], and even take advantage of smart transportation systems [22]. After a recent *E. coli* outbreak in Romaine lettuce, the difficulty in tracing back to the source of infection prompted Walmart Inc. to work with IBM in order to use blockchains storing data of all leafy green vegetables [23]. The technology was also applied in cattle industries. One such example is the Beefchain system in Wyoming that promise the ranch owners with opportunities to acquire the differentiating profits based on the quality of beef (e.g. premium grass-fed) [24].

Different applications have taken different advantages from this technology. Wang *et al.* [25] proposed a decentralized system where smart contracts enable fine grained access control. Blockchains also made possible an anonymous reputation system for vehicular ad hoc networks (VANET) [26]. It has been suggested to keep raw data off chain to deal with scalability issues [10]. The security and the immutability properties of blockchain have found popularity in electronic health record (EHR) systems, a recent work uses interplanetary file system (IPFS) to store data off chain [27]. In supply chains, the benefits can be numerous including proof of product delivery with automatic incentives [28], provenance tracking [29], and traceability [30]. While different applications focus on different benefits of the blockchain, none of them properly fits the requirements of the US animal farming industry. We need a system that can be easily adopted to existing infrastructure while maintaining user anonymity, animal/product traceability, and farm data ownership at the same time.

C. SMART CONTRACTS AND BLOCKCHAIN

Blockchains uses a linked-list type of structure where a block is linked to its previous block via the use of cryptographic hashes. The hashes are computed from the contents of the block. Hence, any modification in the contents of the previous blocks would require all the subsequent blocks to be updated too. This makes it difficult for a perpetrator to change any past data. A block is confirmed in the chain via a process called consensus where the participants in the chain (also called miners) agree on the data contained by the block. There are several configurations for blockchains. Bitcoin is a permissionless and public network where anyone can join and participate. There are frameworks such as Hyperledger or Ethereum that can be deployed in permissioned networks. These can be run in corporate or private setups where only the permitted users can join and participate. There are multiple consensus protocols, the most popular being Proof of Work (PoW). The miner needs to successfully solve a cryptographic problem to mine a block in PoW. This prevents parties with malicious intents who want to corrupt the data, as they will run out of computational resources while going against honest miners in a practical world. However, PoW is computationally expensive and wasteful. In permissioned systems, several other consensus protocols can be used such as Proof of Stake (PoS), Proof of Authority (PoA), etc. These protocols use significantly less computational resources compared to PoW. In PoA based systems, blocks are signed by pre-approved accounts called validators. A block in the chain contains a list of transactions which are analogous to bank checks in the context of cryptocurrencies. Many blockchain frameworks

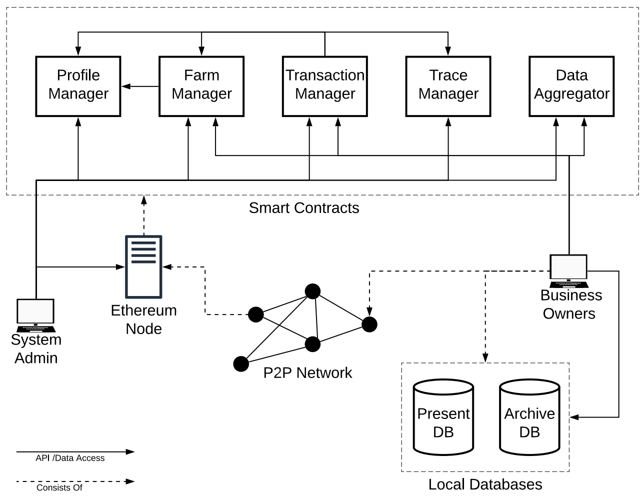


FIGURE 2. A simplified block diagram of the blockchain based farm animal management system. The blockchain network is shown in the bottom where the solid circles represent blockchain nodes (clients). Each blockchain client locally stores smart contracts, associated data, and the respective farm animal databases.

now support smart contracts which are pieces of executable code that run when a block is confirmed and modify the state of the system. Smart contracts can hold codes that can enforce business policies, privacy practices, and access control. They can also hold data. The blockchain framework ensures that everyone has the same version of code and data, which creates trust.

III. THE PROPOSED SYSTEM

We have designed a system which ensures data immutability while preserving data ownership. The system also supports animal traceability, user anonymity, and data aggregation. Our proposed system model is shown in Figure 2. More details about the system can be found in Appendices A and B. Our framework is designed on the Ethereum platform. The framework supports smart contract storage and execution, which is a key to our work. The active nodes in the network communicate as peer-to-peer (P2P), and there is no centralized server like in the client-server model. Outside applications can connect to their respective nearest P2P nodes to retrieve information, manage system operations, or generate transactions. All transactions are stored in each of the active nodes of the network. In addition to the transactions, smart contracts are also stored redundantly in each active node. Cryptographic hashes are computed for both the transactions and the smart contracts, and then stored inside the chain data structure in multiple blocks with timestamps which themselves are linked like a linked list using the cryptographic hashes of their contents. Hence, a modification in the data or the code would require the system to recompute the hashes. As the blockchain data structure is redundantly stored in all the participating nodes, all nodes must agree on the modification (e.g. come to a consensus). We define four major smart contracts for managing the farm animal tracking system: 1) Profile Manager, 2) Farm Manager, 3) Transaction Manager, 4) Trace Manager, and 5) Data Aggregator as shown in Figure 2. The dots in the P2P Network are Ethereum nodes (clients) (Either geth [31] or parity [32] with Proof of Authority (PoA) configuration) where each of them contains Ethereum transaction data, smart contract bytecodes, and the blockchain itself. The business owners can run such nodes in their local systems. Alongside the Ethereum clients, a business owner would also run a local database service for storing their raw farm data. The raw data contain animal data that include inspections, vaccinations, movements and other relevant information. Each time a local database is updated, a cryptographic hash is computed by combining all the raw data of that farm and then stored in the blockchain (via the Farm Manager contract). This hash links the blockchain with the local database. Additional information on the local database operation is available in Appendix B. Privacy and a sense of ownership are the two key reasons why we chose to keep the raw databases local to respective business owners. The system is more resistant to eavesdropping and a business owner is in control of how his/her data are used or shared. However, in order to enforce data immutability and create mutual trust, we use the blockchain to enforce that any alteration in the local data must be reported with an updated hash. Hence, any prospective purchaser or authorized auditing entity can validate whether, how, and when the data have been modified from the original state.

A. ALGORITHMIC PROCEDURES

The proposed blockchain-based animal farm management framework will be able to perform a variety of tasks. We demonstrate some of the basic tasks in this section.

1) USER PROFILE AND FARM MANAGEMENT

Several smart contracts in the system enforce user and farm policies in order to keep the data secure and the identities anonymous. The Ethereum account that deploys a smart contract is automatically assigned as the administrator (admin) for that contract. The admin has certain access permissions to perform tasks related to management. Despite that, the admin and all outside accounts are prevented from accessing and modifying user and farm data. All users of the system (owners, managers, viewers/auditors) must be registered by an admin using the `ProfileManager` contract. All the farms owned by the users must also be registered by the admin using the `FarmManager` before they can participate in the system.

Someone willing to join the network first contacts the network admin. The prospective user provides the admin with a request containing access level desired (`viewer(1)`, `manager(2)`). This is off-chain communication and is completed using traditional methods (emails, messages, letters). Once a request is received and approved, the admin generates a user id which is a 160 bit Ethereum address. The admin registers the user by calling the `registerUser()` method. During this function call, the user ID and the access level is sent as arguments. The `ProfileManager` contract validates whether the function call is coming from an admin

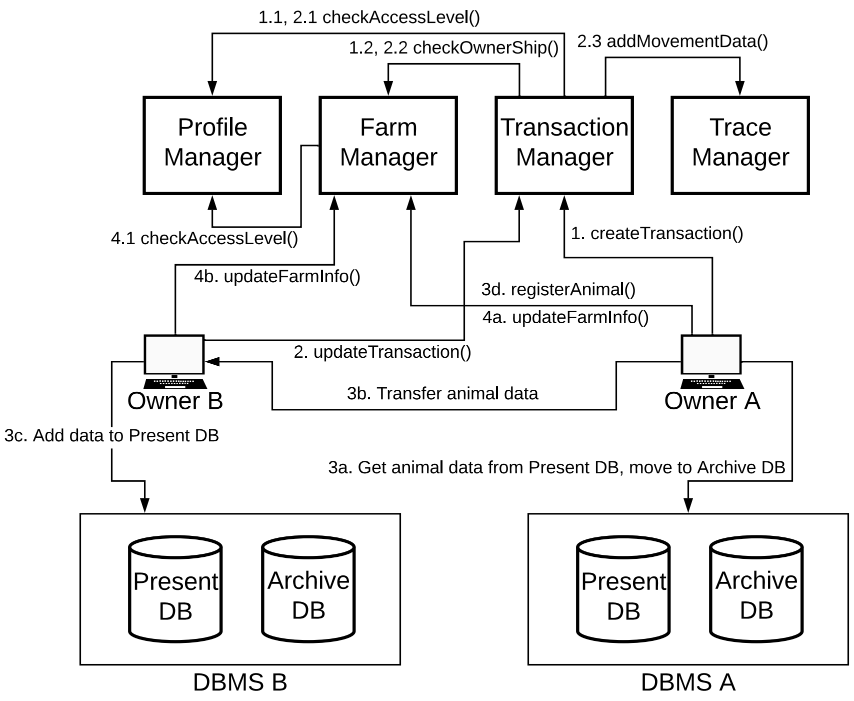


FIGURE 3. Detailed block diagram illustrating communication steps among the system components during a business transaction.

and it checks if the requested user already exists in the system. Upon validation, the Profile Manager registers the user. The admin notifies the user with the confirmation and the users Ethereum address. This address is not shared with anyone else to keep the user identity secret. Once registered, the user can access other contracts of the system. The admin can change user access level by calling the `updateUser()` method. If a user needs to be deactivated/restricted/terminated, the admin can set the access level to 0.

A farm owner willing to register a farm requests the network admin in an off-chain communication. This request contains the owner's Ethereum address that is already registered in the system. The admin generates a farm id which is also a 160 bit Ethereum address. The admin calls the `registerFarm()` method of `FarmManager` contract to register the farm in the system. The `FarmManager` contract validates whether the function call is coming from an admin. It also checks if the farm id already exists in the system or not. Once registered, the `FarmManager` contract contains both the farm id and the owner id in mapped data structure. After the registration is completed, the farm owner can regularly update farm information by calling the `updateFarmInfo()` method of the `FarmManager` contract. During such calls, the animal count and the farm hash are updated by the farm owner.

2) PROCESS OF BUSINESS TRANSACTIONS

A typical example of a financial transaction is shown in Figure 3. We create a typical scenario where two parties (farm owners) A and B exchange some animals. Creating the transaction is a two step process where both owners need to communicate with the `TransactionManager` contract (Steps 1 and 2). In the first step, one farm owner proposes a transaction by calling the `createTransaction()` method. This process creates a `Transaction` object in the contract that contains several information including the addresses of both owners, farm addresses, and the full list of animals being transferred. Animals tags are stored in the list. In the second step, the other owner confirms the transaction using the `updateTransaction()` method. When a farm owner communicates with the `TransactionManager` contract, the contract validates permissions and ownership of the farms between which the transaction would take place. These are marked as sub-steps 1.1, 1.2, 2.1, and 2.2 in Figure 3. Upon confirmation, the `TransactionManager` stores anonymous movement data in the `TraceManager` contract (step 2.3). After the confirmation, the seller/exporter of the animal (Owner A in Figure 3) retrieves the data of the animals in transaction from its local database (step 3a) and transmit directly to the buyer/importer (step 3b). The buyer/importer stores that data in its own local database (step 3c). The seller/exporter registers the sold animals to the new farm (step 3d). The seller also removes the sold animal data from its current database and archives them if necessary. As both the party had their databases modified, they will update the `FarmManager` contract with freshly computed farm hashes (steps 4a and 4b). The `FarmManager` validates their permissions and updates the hash values.

3) ANIMAL TRACING PROCESS

Our framework provides the ability to trace animals using a `TraceManager` contract. Trace data are stored in the contract during a business transaction (See Figure 3, Step 2.3). There can be several reasons why someone would want to trace an animal back to the origins. One such scenario could be when a farm wants to use its reputation to gain prospective customers by giving them a way to securely verify whether their animal/food product came from that farm or not. Another scenario could be tracing source of infection during an outbreak in order to rapidly mitigate the problem. The `TraceManager` contract stores animal movement data in a nested key-value map structure. Each animal is identified by its id (tag) which is used as the key. A custom defined data structure `Animal` is considered to be the value. This structure contains another map listing all the movements, each of which is an instance of the structure `Movement`. We use simple indices as keys to the movement map and keep track of the total number of movements (hence, movement entries). The admin can call the method `getMovementCount()` and provide the animal id in order to know the total number of movement entries. The admin can then call the `getMovementdata()` method with animal id and movement index as arguments in order to get the actual movement entry. A movement entry contains the source farm address, the destination farm address and the time when the movement was recorded in the blockchain.

4) DATA AGGREGATION PROCEDURE

Sometimes the scientific community or the industry can benefit from summary data on animal production. Examples of such data can be average weekly growth of certain breeds, effect of a vaccine, or growth improvements of animal under certain diets. In situations like these, it is needed to provide such information without jeopardising privacy of the business. Hence, we implement a `DataAggregator` contract to help with anonymous data collection. To handle the data collection, the admin first creates a `Dataset` object in the `DataAggregator` contract by calling the `createDataset()` method. The method requires a key and a secret as arguments. If the method succeeds, the key would be used to locate the `Dataset` object and the secret would be used to authenticate the accounts performing read/write operations on the dataset. The admin is responsible to generate, maintain, and distribute this key-secret pair. Using off-chain communications, the admin can request the farm owners to provide certain summary data. These requests contain a description of the data requested and key-secret pair for the dataset. The farm owners, upon receiving such request may or may not decide to participate in the survey. A survey participant calls the `addData()` method to submit data, this method requires the actual data, the key, and the secret as its arguments. Once the submission period ends, the admin may obtain the stored data using the `getData()` method which also requires the key and the secret. Only the farms participating in the data sharing will be able to access the aggregated information.

IV. SYSTEM ANALYSIS

In this section we evaluate our system and explain how it handles anonymity of users, data security, and several other aspects of trust.

A. USER PRIVACY

The private Ethereum blockchain users can operate in the system without providing information that can identify them or their location. Each user is provisioned by the system administrator when an externally owned account (EOA) in Ethereum is generated. These accounts have several components including private keys and Ethereum addresses. Private keys are always securely stored by the user itself and never revealed outside. The public keys are derived from private keys. The addresses are derived from public keys using the Keccak-256 hash function where last 20 bytes (LSB) of the hash are kept. The one-way hash function prevents association back to the public key. Hence, even after knowing someone's address, one cannot derive the identity of that person.

Due to the system configuration as a private blockchain network, outsiders generally cannot enter the system without proper authorization from the administrator. If someone already in the system, wants to impersonate another user, it would be automatically prevented if private keys are not accessible by anyone other than the legitimate owner.

B. DATA SECURITY

Each farm business may choose to store animal-related data locally in their premises. A locally running relational database (SQL) management system would contain tables of data as shown in Figure 14. Each time the tables are updated, a cryptographic hash is generated for each table using a hash function. Eventually, all the table level hashes are combined to generate a single hash which is stored in the `FarmManager` contract by the user. This technique ensures that data cannot be altered by single owner without updating the hash in the Ethereum blockchain. If one needs to verify the integrity of data, he/she can do so by recalculating the hashes from the data. To enable faster verification of data, the `FarmManager` contract also stores in its state variables, hashes related to individual animal data. When animals are transferred during a transaction, animal data are sent to the new owner (steps 3a, 3b, and 3c in Figure 3). The new owner can check the animal hashes stored in the `FarmManager` contract and verify if they match with the data by recomputing hashes from the received data.

C. PROVENANCE

The `TraceManager` contract stores trace data for every animal that is moved from one farm to another. In addition to that, every farm owner also keeps a record of the past movement history of an animal with cryptographic hash as proof in the blockchain system. While the cryptographic hashes ensure the integrity of the data, the movement records in the `TraceManager` contract provides faster tracing without requesting data from independent local databases. If someone wants to fabricate the origin of an animal or animal product, he/she would face two major obstacles: i) the hashes stored for the animal movement data won't match and ii) the trace data captured by the `TraceManager` would not match. As a hypothetical situation, let's assume the person with an intent of corrupting animal trace data forms a coalition with all the owners involved in the animal transfer process. As the local databases are controlled by their respective owners, they could theoretically store and exchange fabricated movement data and make it look real. However, the `TraceManager` prevents such fabrication by making the tracing process automatic. It only captures data from the `TransactionManager` and for that reason, the entities (owners and farms) involved in a transaction cannot fabricate their addresses as they can only authenticate with their own IDs.

D. SECURED DATA AGGREGATION

While surveys and collections of data are mostly beneficial, owners may still be discouraged from a business perspective if they fear loss of privacy. Our system contains a smart contract in order to facilitate anonymous data collection. It uses a simple key-secret pair to authenticate a submitter during data collection and does not use any other identification mechanism that can be exploited by others. Let's assume a hypothetical scenario, where the admin itself is corrupted in its ability to protect user privacy. Although the admin has a basic idea of who a user is due to the management of user profile and farm ids, it does not have any access to user or farm data. If the surveys were conducted by the admin in a direct manner to collect user data, it could identify who sent the data. However, the `DataAggregator` prevents this issue by not keeping records of survey participants. The admin, who has access to aggregated data, can only know how many entries were submitted and what was submitted.

E. FAIRNESS OF THE SYSTEM

The system provides comparable levels of accessibility, security, and privacy to its users. The consensus mechanism does not differentiate among the participants. The lack of central authority eliminates bias and gives power back to the individual entities. The local data ownership mechanisms indicated in Figure 2 give the users more control over their private business data. The origin tracking (provenance) makes the system fair for the consumers as well who lie at the end of the chain. Consumers can verify the authenticity of claims about beef products made by the business entities.

F. RELIABILITY

Traditional server-centric systems have a vulnerability commonly referred to as: single point of failure. The redundancy introduced by the proposed framework which takes advantage of the Ethereum blockchain can mitigate this issue. We categorize failures into two types: link failure and node failure. A link failure is defined as the situation when a link connecting two blockchain nodes is unable to sustain communication. On the other hand, a node failure is defined as the scenario when a node cannot communicate with the network via any of its links or any data stored on the node is lost. As we are dealing with a peer to peer (P2P) network, link failures do not affect operations unless enough links fail to disconnect or isolate a node. A fully connected network with n nodes has $n(n-1)/2$ links (maximum number of edges in an undirected graph). To keep the network fully operational we need at least $n-1$ links (minimum number edges needed to maintain connectivity in an undirected graph). Hence, up to $(n-1)(n-2)/2$ links can fail, and the system can still operate. Note, this number depends on which links are failing. In worse cases when all links connecting to a node fail, that node becomes isolated. A user may still connect to the network via any alternate node as the credentials are valid across the system. Once the failed links recover, the node can come out of isolation and re-sync all data. A node failure has the same effect as the complete link failure described above. If data is lost or damaged due to a node failure, the node can re-sync once connection is re-established. Data loss is tolerable up to $n-1$ nodes as every node contains all the blockchain related data and smart contract bytecodes.

G. COMPUTATIONAL COSTS

In Ethereum, gas cost is a measure of how much computational resource or storage is needed to complete a transaction or a smart contract operation. Any operation that changes the state of the Ethereum virtual machine (EVM) creates a transaction and every such transaction has an associated gas cost. Although gas costs imply spending real money in the form of Ether when the main Ethereum network is used, it is not the same in our case. In private chains, Ether has no value. Nevertheless, it is an available measure of system resource usage. We demonstrate the gas costs of contract creation in Figure 4 and the costs of calling some commonly used methods in Figure 5. In these figures, a higher gas cost indicate that more processor cycles, dynamic memory, or persistent storage are required in order to complete an operation. It is important to note that, read only methods (that does not change any data in the memory or the state of the chain) do not create transactions at all. As every node has a snapshot of the system, such methods simply read data from the local running node.

The `FarmManager` and the `TransactionManager` contracts implement a large number of functionalities (See Figures 10 and 11), hence, it is no wonder that, they cost a lot more gas (about three times more) compared to others. The contract deployments are one-time operations which are done at the beginning of the system setup. The operations depicted in Figure 5 can occur at any arbitrary time once the system is running. These methods change the state of the system by adding or modifying data in both memory and persistent storage. The methods related to business transactions and movement data consume more resources

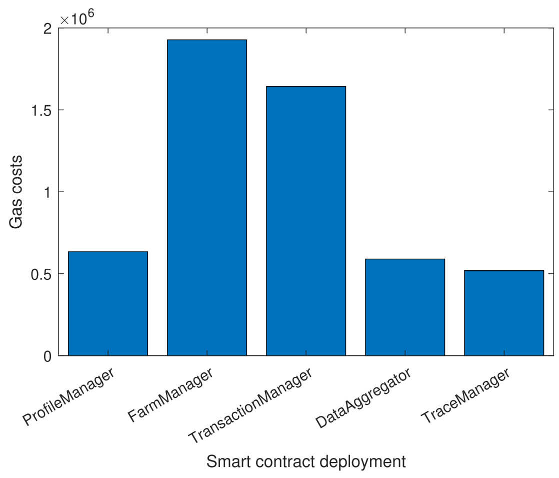


FIGURE 4. **A comparative chart showing contract creation (deployment) costs for the five proposed smart contracts. Gas cost is a measure of computational resources (processing power and storage) needed for the operation.**

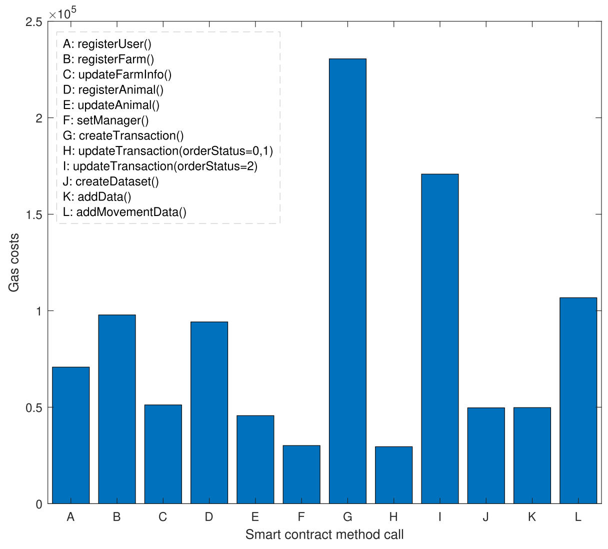
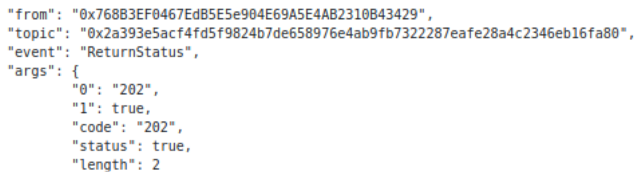


FIGURE 5. **A comparative chart showing costs of calling different smart contract methods (functions) that change the state of the system. Read only (view) methods are not included here. Gas cost is a measure of computational resources (processing power and storage) needed for the operation.**

mostly due to their space complexity. They create new objects and add new information. The `updateTransaction()` method has two distinct scenarios. When a transaction is `confirmed` (`orderStatus = 2`), it stores the movement data (internally calls the `addMovementData()` method). Hence, it requires significantly more gas compared to the other scenarios such as, an order being `proposed` (`orderStatus = 1`).

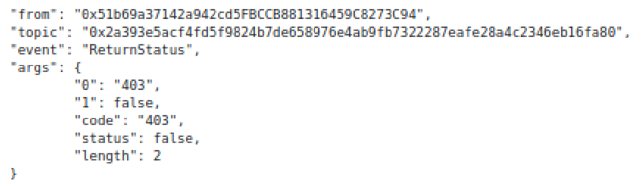
H. INTEGRATION TEST

We test several operational scenarios with a prototype running system. The configuration of the prototype where these scenarios are simulated and tested is described in the appendix.



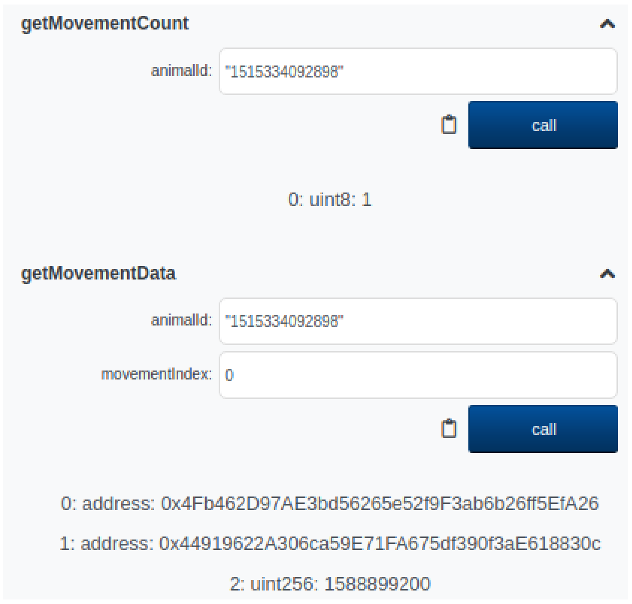
```
"from": "0x768B3EF0467EdB5E5e904E69A5E4AB2310B43429",
"topic": "0x2a393e5acf4fd5f9824b7de658976e4ab9fb7322287eafe28a4c2346eb16fa80",
"event": "ReturnStatus",
"args": {
        "0": "202",
        "1": true,
        "code": "202",
        "status": true,
        "length": 2
```

FIGURE 6. **The logs generated as a response by the `FarmManager` (with address `0x768B...3429`) contract when an authorized farm owner attempts to update farm info. This result is generated from test case 1 of Table 1.**



```
"from": "0x51b69a37142a942cd5FBCCB881316459C8273C94",
"topic": "0x2a393e5acf4fd5f9824b7de658976e4ab9fb7322287eafe28a4c2346eb16fa80",
"event": "ReturnStatus",
"args": {
        "0": "403",
        "1": false,
        "code": "403",
        "status": false,
        "length": 2
}
```

FIGURE 7. **The logs generated as a response by the `TransactionManager` (with address `0x51B6...3C94`) contract when an unregistered user attempts to create a business transaction. This result is generated from test case 5 of Table 1.**



animalId: "1515334092898"

0: uint8: 1

animalId: "1515334092898"

movementIndex: 0

0: address: 0x4Fb462D97AE3bd56265e52f9F3ab6b26ff5EfA26

1: address: 0x44919622A306ca59E71FA675df390f3aE618830c

2: uint256: 1588899200

FIGURE 8. **The returned outputs when the authorized admin attempts to read trace data by calling the methods provided by the `TraceManager` contract. Here, the results indicate an animal with id `1515334092898` was moved from farm A (`0x4FB4...FA26`) to farm B (`0x4491...830C`). This result is generated from test case 6 of Table 1.**

Every scenario consists of doing some tasks in order to simulate a situation. These test scenarios along with the outcomes are described in Table 1. Every case is classified as either positive or negative. The positive cases are those where all the communications done with system are expected and valid by design. The negative cases are those where one or more operations done with the system are defined as illegal or there was a failure in authentication or validation. The system is designed in a way to provide secured environment and ensure any breach of data or operational access. The logs and results generated for few of the test cases are shown

TABLE 1. Integration test procedures and results.

No.	Test Case	Test type	Prerequisites	Tasks	Results
1	Authorized farm owner attempts to update farm info	Positive	i) Admin registers the farm owner (node 2) with access level 2. ii) Admin registers the farm (farm A) and appoints ownership (farm A - node 2) to the appropriate owner.	The farm owner (node 2) calls `updateFarmInfo()` to update farm (farm A) information.	Status code 202. The `FarmManager` contract updates the information when the next block is mined.
2	Unregistered user attempts to update farm info	Negative	The user (node 4) is never registered to the system.	Unregistered user (node 4) calls `updateFarmInfo()` to update farm (farm A) information.	Status code 403. The `FarmManager` contract rejects the request as the validation fails.
3	Registered user attempts to update info of a farm that is not owned	Negative	i) The user (node 3) is registered with access level 2. ii) The farm (farm A) is never assigned as owned by the user (node 3).	The user (node 3) calls `updateFarmInfo()` to update farm info for the farm (farm A) that is not owned by it.	Status code 403. The `FarmManager` contract rejects the request as the validation fails.
4	Authorized farm owner attempts to create a business transaction with another authorized owner	Positive	i) Admin registers the users (node 2 and node 3) with access level 2. ii) Admin registers the farms (farm A and farm B) and appoints ownership (node 2 - farm A and node 3 - farm B) to the respective users.	One farm owner (node 2) calls `createTransaction()` to create a business transaction of several animals to be transferred from one farm (farm A) to another farm (farm B).	Status code 202. The `TransactionManager` contract accepts the proposal upon validation and registers a transaction when the next block is mined.
5	Unregistered user attempts to create a business transaction	Negative	The user (node 4) is never registered to the system.	Unregistered user (node 4) calls `createTransaction()` to create a business transaction.	Status code 403. The `TransactionManager` contract rejects the request as the validation fails.
6	Admin attempts to retrieve trace data	Positive	At least one transaction must be created and confirmed. Note: Here the transaction 1581115549157 moved the animals [1515334092888,1515334092898] from farm A (`0x4FB4...FA26`) to farm B (`0x4491...830C`).	i) The admin calls the `getMovementCount()` method to know the number of movements recorded for an animal (1515334092898). ii) The admin calls the `getMovementData()` method to get each row of the movement entries for an animal (1515334092898)	The `TraceManager` contract responds to the method calls and returns the results.
7	Unauthorized user attempts to retrieve trace data	Negative	At least a single transaction must be created and confirmed. Note: Here the transaction 1581115549157 moved the animals [1515334092888,1515334092898] from farm A (`0x4FB4...FA26`) to farm B (`0x4491...830C`).	i) Any user who is not an admin (node 4) calls the `getMovementCount()` method to know the number of movements recorded for an animal (1515334092888). ii) Any user who is not an admin (node 4) calls the `getMovementData()` method to get each row of the movement entries for an animal (1515334092888),	The `TraceManager` contract responds with null value (0) as the validations fail.
8	Invited user attempts to submit survey data	Positive	i) Admin registers a data set with a key and a secret. (We use key: 1001, secret: 2020) ii) The key and the secret are mailed to every survey participant.	An user who is a participant (node 4) calls the `addData()` method to submit some data to the data set.	Status code 202. The `DataAggregator` contract accepts the submission upon validation and stores it when the next block is mined.
9	Unauthorized user attempts to submit survey data	Negative	i) Admin registers a data set with a key and a secret. (We use key: 1001, secret: 2020) ii) The test user gets the incorrect key and secret (1003, 2019).	An uninvited user (node 3) calls `addData()` method with wrong key-secret pair to submit some data to the data set.	Status code 403. The `DataAggregator` contract rejects the submission upon validation.
10	Admin attempts to retrieve summary data	Positive	A data set must be registered and at least a single participant must submit some data. Note: Here the data set with key: 1001 and secret: 2020 is used. A user (node 4) submitted some data (0x44) in the survey.	i) The admin calls the `getDataCount()` method to know the number of data entries recorded in the survey. ii) The admin calls the `getData()` method to get each row of data collected in the survey.	The `DataAggregator` contract responds to the method calls and returns the results.

in Figures 6, 7, and 8. The complete set of results can be found in the supplementary materials document. The table validates how our proposed system is temper proof and provides data security and traceability while maintaining user anonymity.

V. CONCLUSION

In this article, we proposed a blockchain based supply chain management framework to be used in the US beef cattle industry. We explained in detail how this system will operate and communicate with various entities involved. Finally, we analyzed how the framework will work in order to ensure user anonymity, improve data privacy, and ensure trace data integrity. We performed integration tests to evaluate the system operation in various scenarios.

The proposed framework operates as a private / consortium blockchain and uses proof of authority (PoA) for achieving consensus, eliminating the computationally expensive hash computations. This enables us to run complex smart contract functions and store more data on smart contract to enable traceability and improve data security. The database can be hosted on any SQL database management system by properly configuring the specified schema. The smart contracts can be run on any Ethereum based clients (geth, parity etc.). The system requires an admin who will initiate everything and register the users and their farm businesses. This could be a potential weak point as the admin will know the mappings of the user and the farm addresses. Hence, despite being a trustless decentralized architecture dealing with supply chains, the framework requires some trust. However, the smart contracts are carefully designed to isolate private data from admins and any unintended users. The owners do have control on who can see their data. Different designs in the architecture may affect privacy, security, and resource requirements. This design is optimized for the US beef cattle industry with a focus on privacy, data ownership, and security. The system runs optimally with traditional computational resources and does not require any special hardware. The technical knowledge on blockchains is also commonplace given that it is being used in varying types of industries including cryptocurrenices. However, blockchains do have scalability issues when it comes to storage. With time, the requirements on persistent storage would increase. Ethereum supports light client nodes to help with such an issue. In the future, it may also be possible for the blockchain systems to discard data old enough to become irrelevant. Blockchains also face adaptability issues as an emerging technology and requires its users to go through a learning curve.

Our proposed system provides a technical solution to several specific issues encountered by the US farm industry. It improves over the existing knowledge on how blockchains can be utilized to meet the specific industrial needs such as identity protection and data ownership while ensuring immutability and product traceability. The framework can be used in other supply chains with minor modifications. Future work can focus on improving scalability of blockchains in general. While different industries have varying requirements, most can benefit from scalable, secured, and efficient blockchain frameworks.

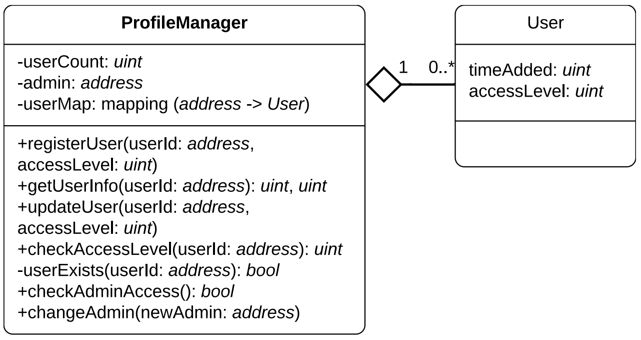

ProfileManager
-userCount: *uint*
-admin: *address*
-userMap: mapping (*address* -> *User*)
+registerUser(userId: *address*, accessLevel: *uint*)
+getUserInfo(userId: *address*): *uint*, *uint*
+updateUser(userId: *address*, accessLevel: *uint*)
+checkAccessLevel(userId: *address*): *uint*
-userExists(userId: *address*): *bool*
+checkAdminAccess(): *bool*
+changeAdmin(newAdmin: *address*)
1
0..*
User
timeAdded: *uint*
accessLevel: *uint*

FIGURE 9. The `ProfileManager` contract and its associated `User` data structure. There can be multiple user profiles in the system, all of which must be registered via the methods of this contract.

APPENDIX A
SMART CONTRACTS

The smart contracts are immutable pieces of code that runs in the blockchain system. Our system has five smart contracts, each with different objectives. In each of those, we define variables, objects, and methods to enable different management capabilities. The methods are designed to enforce policies regarding how to handle different aspects of the system including permissions, data access etc. All of these contracts are loaded into the system and configured by an administrator during the system initialization process. In response to different method calls, we use Ethereum event logs to understand different outcomes. We use the following HTTP style response codes in these logs: 200 (OK/success), 201 (created), 202 (accepted), 400 (bad request), 403 (forbidden), and 404 (not found).

A. PROFILE MANAGER

Every entity that accesses the system is managed and controlled by the Profile Manager. As shown in Figure 9, we define a data structure `User`, which has the following components:

- `timeAdded` is the unix timestamp when the user profile is created.
- `accessLevel` is the access classification for the user. There are 3 distinct access levels: `restricted (0)`, `viewer (1)`, and `manager (2)`.

Each entry of the `User` data structure is mapped through a 160 bit Ethereum `address` which is the user id. The `ProfileManager` contract also contains several state variables:

- `userCount` is the number of registered users
- `admin` is the address of the system administrator.
- `userMap` is the mapping data structure that maps user id to `User` object.

Initially, the creator of the smart contract is automatically added as the admin by the constructor function. Only the admin can register new user profiles, change permissions, and change admins. Most of the methods (functions) contain code snippets that validate the entity that invoked the call. The methods in this contract are listed below,

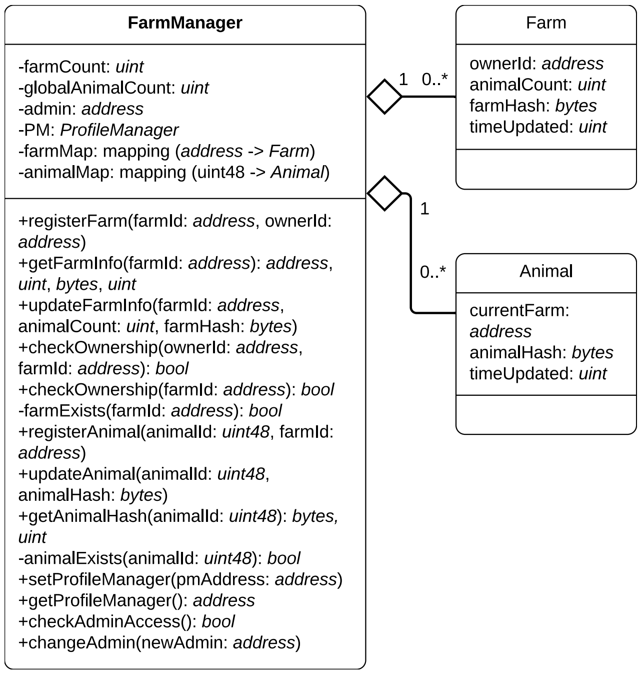

FarmManager
-farmCount: *uint*
-globalAnimalCount: *uint*
-admin: *address*
-PM: *ProfileManager*
-farmMap: mapping (*address* -> *Farm*)
-animalMap: mapping (uint48 -> *Animal*)
+registerFarm(farmId: *address*, ownerId: *address*)
+getFarmInfo(farmId: *address*): *address*, *uint*, *bytes*, *uint*
+updateFarmInfo(farmId: *address*, animalCount: *uint*, farmHash: *bytes*)
+checkOwnership(ownerId: *address*, farmId: *address*): *bool*
+checkOwnership(farmId: *address*): *bool*
-farmExists(farmId: *address*): *bool*
+registerAnimal(animalId: *uint48*, farmId: *address*)
+updateAnimal(animalId: *uint48*, animalHash: *bytes*)
+getAnimalHash(animalId: *uint48*): *bytes*, *uint*
-animalExists(animalId: *uint48*): *bool*
+setProfileManager(pmAddress: *address*)
+getProfileManager(): *address*
+checkAdminAccess(): *bool*
+changeAdmin(newAdmin: *address*)
1
0..*
Farm
ownerId: *address*
animalCount: *uint*
farmHash: *bytes*
timeUpdated: *uint*
1
0..*
Animal
currentFarm: *address*
animalHash: *bytes*
timeUpdated: *uint*

FIGURE 10. **The `FarmManager` contract and its associated `Farm` and `Animal` data structures.**

- `registerUser()` adds a new profile to the system, creates a `User` object and stores it using the `userMap` mapping. Only an admin can call this function.
- `getUserInfo()` returns information about a user based on the address given in the argument. The requesting entity must be an admin or the user itself.
- `updateUser()` updates the user profile. An admin can call this function to change access permissions.
- `checkAccessLevel()` returns the access level of a user given its address in the argument.
- `checkAdminAccess()` checks whether the current contract method calling entity has the admin level access (i.e., is the admin) or not.
- `changeAdmin()` assigns a new admin given the `address` in the argument. Only the current admin can call this function. Once called, if the contract transaction is confirmed, the current admin will lose its status as admin.

B. FARM MANAGER

The Farm Manager contract regulates the contents of the farm databases (marked as 'Local Databases' in Figure 2) and provides useful farm related operational functionalities. The contract and its associated data structures are shown in Figure 10. To store cryptographic proof of farm database contents, we define the `Farm` data structure which has the following components:

- `ownerId` is the 160 bit Ethereum `address` of the farm owners profile.
- `animalCount` is the number of animals currently registered in the farm.
- `farmHash` is the most recent cryptographic hash generated from the local database of the farm.
- `timeUpdated` is the unix timestamp when the most recent `farmHash` was stored in the chain.

In addition to that, we define the `Animal` data structure with the following components to store cryptographic proofs of individual animal related data:

- `currentFarm` is the 160 bit Ethereum `address` of the farm that owns the animal.
- `animalHash` is the most recent cryptographic hash generated from the animal from its information stored in the database.
- `timeUpdated` is the unix timestamp when the most recent `animalHash` was stored in the chain.

Each entry of the `Farm` data structure is mapped through a 160 bit Ethereum `address` which is the farm id. Each entry of the `Animal` data structure is mapped through a 48 bit animal id number. The `FarmManager` contract contains several state variables:

- `farmCount` is the number of registered farms.
- `globalAnimalCount` is the total number of animals combining all the registered farms.
- `admin` is the 160 bit Ethereum `address` of the system administrator.
- `PM` is an object containing the address of the `ProfileManager` contract. The methods defined in `ProfileManager` can be used to validate user privileges (For example, checking user access level).
- `farmMap` is the mapping data structure that maps farm id to `Farm` object.
- `animalMap` is the mapping data structure that maps animal id to `Animal` object.

Initially, the creator of the smart contract is automatically added as the admin by the constructor function. The admin can register new farms, link `ProfileManager` contract deployed in the chain, and change admins. However, the admin cannot access the contents of `Farm` data objects, only the registered farm owners (i.e., `Farm.ownerId`) can do so. The methods (functions) described below contain code snippets to enforce such access control. The methods in this contract are listed below:

- `registerFarm()` adds a farm to the system, creates a `Farm` object and stores it in the contract. Only an admin can call this function.
- `getFarmInfo()` returns information about a farm based on the farm id (`address`) provided in the argument. Only a farm owner with proper access level can call this function.
- `updateFarmInfo()` updates information about the farm. Only a farm owner with proper access level can call this function.
- `checkOwnership()` validates if the owner referenced by the `ownerId` in the argument or the method calling entity owns the farm referenced by the `farmId` in the argument. It's an overloaded method.

- `farmExists()` checks whether a given farm referenced by the `farmId` in the argument exists or not.
- `registerAnimal()` adds an animal to the system, creates an `Animal` object and stores it in the contract. If the animal is already registered, the method updates the id of the farm that the animal is in.
- `updateAnimal()` updates the cryptographic hash of an animal. Only the owner of the farm that contains the animal can call this method.
- `getAnimalHash()` returns the cryptographic proof of the animal data that is stored in the contract.
- `animalExists()` checks whether a given animal referenced by the `animalId` in the argument exists or not.
- `setProfileManager()` instantiates the `PM` (i.e., `ProfileManager`) object with the Ethereum `address` of the Profile Manager contract deployed in the chain. Only an admin can call this function.
- `getProfileManager()` returns the address of the Profile Manager contract which is linked to this contract. Only an admin can call this function.
- `checkAdminAccess()` checks whether the current contract method calling entity has the admin level access (i.e., is the admin) or not.
- `changeAdmin()` assigns a new admin given the `address` in the argument. Only the current admin can call this function. Once called, if the contract transaction is confirmed, the current admin will lose its status as admin.

C. TRANSACTION MANAGER

This contract handles business transactions that result in transfers of animals among farms. It provides methods for both the parties (sender and recipient) to initiate and confirm transactions. It also automatically calls appropriate methods of `TraceManager` to store movement data. The contract and its associated `Transaction` data structure are shown in Figure 11. The `Transaction` data structure has the following components:

- `srcFarm` is the 160 bit Ethereum `address` of the source farm (id).
- `dstFarm` is the 160 bit Ethereum `address` of the destination farm (id).
- `srcOwner` is the 160 bit Ethereum `address` of the owner (id) of the source farm.
- `dstOwner` is the 160 bit Ethereum `address` of the owner (id) of the destination farm.
- `orderStatus` is the current state of the order. It can be either of the following values: `proposed (1)`, `confirmed (2)`, or `canceled (3)`.
- `animalCount` is the number of animals listed in this transaction.
- `animalMap` is the mapping data structure that stores the tags of the animals listed in the transaction. An auto incrementing index is used as the key which goes from `0` to `animalCount - 1`.

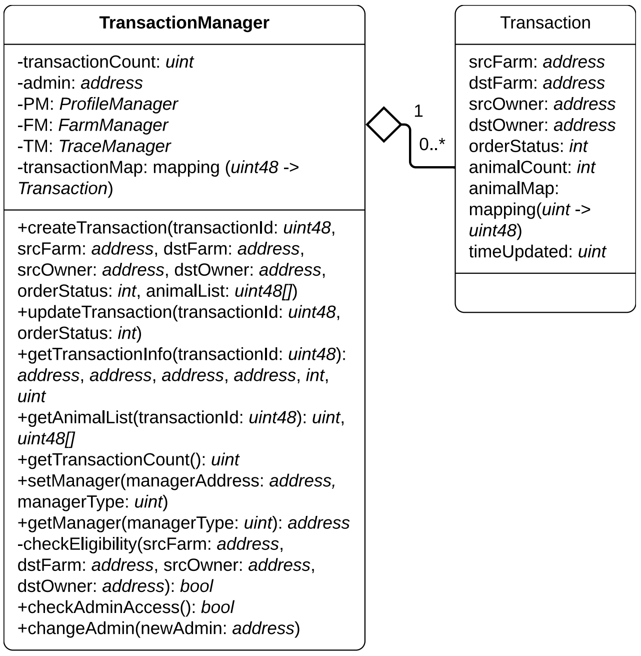

TransactionManager
-transactionCount: *uint*
-admin: *address*
-PM: *ProfileManager*
-FM: *FarmManager*
-TM: *TraceManager*
-transactionMap: mapping (*uint48* -> *Transaction*)
+createTransaction(transactionId: *uint48*, srcFarm: *address*, dstFarm: *address*, srcOwner: *address*, dstOwner: *address*, orderStatus: *int*, animalList: *uint48[]*)
+updateTransaction(transactionId: *uint48*, orderStatus: *int*)
+getTransactionInfo(transactionId: *uint48*): *address*, *address*, *address*, *address*, *int*, *uint*
+getAnimalList(transactionId: *uint48*): *uint*, *uint48[]*
+getTransactionCount(): *uint*
+setManager(managerAddress: *address*, managerType: *uint*)
+getManager(managerType: *uint*): *address*
-checkEligibility(srcFarm: *address*, dstFarm: *address*, srcOwner: *address*, dstOwner: *address*): *bool*
+checkAdminAccess(): *bool*
+changeAdmin(newAdmin: *address*)
1
0..*
Transaction
srcFarm: *address*
dstFarm: *address*
srcOwner: *address*
dstOwner: *address*
orderStatus: *int*
animalCount: *int*
animalMap: mapping(*uint* -> *uint48*)
timeUpdated: *uint*

FIGURE 11. The `TransactionManager` contract and its associated `Transaction` data structure.

- `timeUpdated` is the most recent unix timestamp when the transaction was created/modified.

Each entry of the `Transaction` object is mapped through a 48 bit transaction id which is the unix timestamp of when the transaction was generated by a client. The `TransactionManager` contract also contains several state variables:

- `transactionCount` is the number of transactions handled by the manager so far.
- `admin` is the address of the system administrator.
- `PM` is an object containing the address of the `ProfileManager` contract. The methods defined in `ProfileManager` can be used to validate user privileges (For example, checking user access level).
- `FM` is an object containing the address of the `FarmManager` contract. The methods defined in `FarmManager` can be used to validate the ownership of the farms (For example, checking if user A owns farm F).
- `TM` is an object containing the address of the `TraceManager` contract. The methods defined in `TraceManager` can be used to store animal movement data.
- `transactionMap` is the mapping data structure that maps transaction id to `Transaction` object.

Initially, the creator of the smart contract is automatically added as the admin by the constructor function. The admin can link other contracts such as, `ProfileManager`, `FarmManager`, and `TraceManager` deployed in the chain and change admins. However, admins cannot create, update, or access business transaction data, only the parties involved in the transaction can do so. The methods in this contract are listed below:

- `createTransaction()` creates a new transaction and submits it into the system to be processed by all the participants listed in the transaction. The method calling entity must be an owner of one of the farms involved in the transaction with necessary privileges.
- `updateTransaction()` updates the state of an existing transaction. The method calling entity must be an owner of one of the farms involved in the transaction with necessary privileges.
- `getTransactionInfo()` returns the contents of an existing transaction stored in the contract. The method calling entity must be either the `srcOwner` or the `dstOwner` listed in the transaction.
- `getAnimalList()` returns an array containing the tags of the animals listed in the transaction. The method calling entity must be either the `srcOwner` or the `dstOwner` listed in the transaction.
- `getTransactionCount()` returns the total number of transactions handled by the `TransactionManager` so far.
- `setManager()` instantiates one of the three manager contract (`PM, FM, TM`) objects with the Ethereum `address` of that respective contract deployed in the chain. The argument `managerType` determines which manager to instantiate (`ProfileManager (1)`, `FarmManager (2)`, or `TraceManager (3)`). Only an admin can call this function.
- `getManager()` returns the address of one of the manager contracts which is linked to this contract. The argument `managerType` determines which manager the query is about. Only an admin can call this function.
- `checkEligibility()` is a private helper method that validates the ownership of farms (by calling a `FarmManager` method) and access privileges (by calling a `ProfileManager` method) of users. This is used by other methods in this contract.
- `checkAdminAccess()` checks whether the current contract method calling entity has the admin level access (i.e., is the admin) or not.
- `changeAdmin()` assigns a new admin given the `address` in the argument. Only the current admin can call this function. Once called, if the contract transaction is confirmed, the current admin will lose its status as admin.

D. TRACE MANAGER

This contract enables traceability in the supply chain. It provides methods that the admin can use in urgent situations to trace movements of targeted animals. The contract and its associated data structure are shown in Figure 12. The `Animal` structure contains all the movement data encapsulated using the `Movement` structure which has the following components:

- `srcFarm` is the id of farm that has sold/ delivered the animal.
- `dstFarm` is the id of farm that has purchased/ received the animal.
- `timeMoved` is the unix timestamp when the transfer (transaction) was confirmed.

Each entry of the `Movement` object is mapped through an 8 bit auto-generated index in the `Animal` object. The `Animal` structure contains the following components:

- `movementMap` is the mapping data structure that maps an unsigned integer index to `Movement` object. The index is handled automatically and it is local to each `Animal` object.
- `movementCount` is the total number of movement entries for an animal.

Each entry of the `Animal` object is mapped through a 48 bit animal id (tag) in the `TraceManager` contract. The `TraceManager` contract contains the following state variables:

- `animalCount` is the number of animals handled by the Trace Manager so far.
- `admin` is the address of the system administrator.
- `animalMap` is the mapping data structure that maps animal id (tag) to `Animal` object.

Initially, the creator of the smart contract is automatically added as the admin by the constructor function. The `TransactionManager` contract automatically uses methods from this contract in order to add/update trace data. However, only the admin can inquire this contract about movement data on a particular animal. The methods in this contract are listed below:

- `addMovementData()` adds a single entry of movement data for a particular animal.
- `getMovementCount()` returns the total number of movements that were recorded for a particular animal. Only an admin can call this method.
- `getMovementData()` returns a single entry of movement data for a particular animal given the animal id and the index of the movement. Only an admin can call this method.
- `checkAdminAccess()` checks whether the current contract method calling entity has the admin level access (i.e., is the admin) or not.
- `changeAdmin()` assigns a new admin given the `address` in the argument. Only the current admin can call this method. Once called, if the contract transaction is confirmed, the current admin will lose its status as admin.

E. DATA AGGREGATOR

This contract provides data structures and methods using which the network administrator can collect and manage anonymous survey data on the farming industry. The contract and its associated `Dataset` structure are shown in Figure 13. The `Dataset` structure has the following components:

- `dataMap` is the mapping data structure that maps an integer index to `byte` data. The index is handled automatically and it is local to each `Dataset` object.
- `dataCount` is the total number of data entries for a `Dataset`.

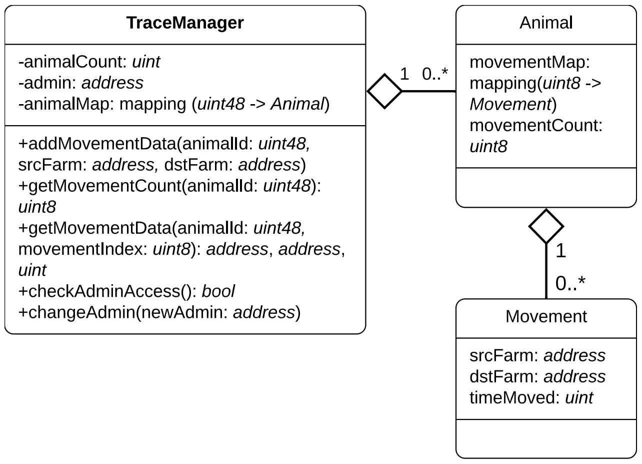

TraceManager
-animalCount: uint
-admin: address
-animalMap: mapping (uint48 -> Animal)
+addMovementData(animalId: uint48, srcFarm: address, dstFarm: address)
+getMovementCount(animalId: uint48): uint8
+getMovementData(animalId: uint48, movementIndex: uint8): address, address, uint
+checkAdminAccess(): bool
+changeAdmin(newAdmin: address)
1
0..*
Animal
movementMap: mapping(uint8 -> Movement)
movementCount: uint8
1
0..*
Movement
srcFarm: address
dstFarm: address
timeMoved: uint

FIGURE 12. **The `TraceManager` contract and its associated `Animal` and `Movement` data structures.**

Each entry of the `Dataset` structure is mapped through an unsigned integer which is regarded as the key of the data set. In addition to the `Dataset` structure, the data aggregator has the following state variables:

- `datasetCount` is the number of data sets handled by the aggregator so far.
- `admin` is the address of the system administrator.
- `datasetMap` is the mapping data structure that maps an integer key to a `Dataset` object.
- `secretMap` is the mapping data structure that maps an integer key to a confidential access code / password (regarded as the secret).

Initially, the creator of the smart contract is automatically added as the admin by the constructor method. The admin can add and configure new datasets, associate `DataSet` keys with secrets, and retrieve stored data. The users can add entries in a dataset if they can validate with the correct key-secret pair. This contract has the following methods:

- `addData()` adds a single entry of `byte` data for a particular dataset. The key and the secret given in the argument determines the set and authenticates the entry.
- `createDataset()` creates and configures a new `Dataset` object. Can only be called by the admin. The key and the secret provided in the arguments must match in the future for user entries.
- `getDataCount()` returns the total number of entries that were recorded for a particular data set. Only an admin can call this function.
- `getData()` returns a single entry of data for a particular dataset given the key, secret, and the data index. Only an admin can call this function.
- `checkAdminAccess()` checks whether the current contract method calling entity has the admin level access (i.e., is the admin) or not.
- `changeAdmin()` assigns a new admin given the `address` in the argument. Only the current admin can call this method. Once called, if the contract transaction is confirmed, the current admin will lose its status as admin.

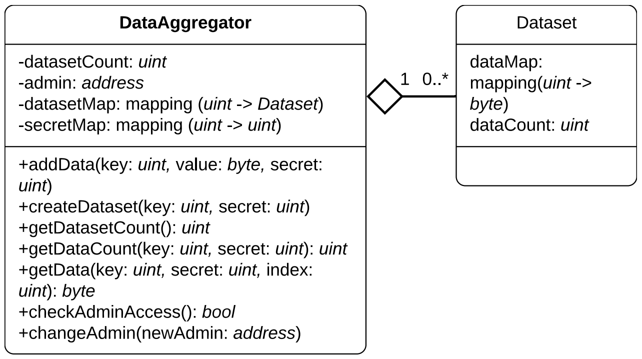

DataAggregator
-datasetCount: uint
-admin: address
-datasetMap: mapping (uint -> Dataset)
-secretMap: mapping (uint -> uint)
+addData(key: uint, value: byte, secret: uint)
+createDataset(key: uint, secret: uint)
+getDatasetCount(): uint
+getDataCount(key: uint, secret: uint): uint
+getData(key: uint, secret: uint, index: uint): byte
+checkAdminAccess(): bool
+changeAdmin(newAdmin: address)
1
0..*
Dataset
dataMap: mapping(uint -> byte)
dataCount: uint

FIGURE 13. **The `DataAggregator` contract and its associated `Dataset` data structures.**

APPENDIX B
FARM ANIMAL DATABASE

The blockchain nodes store the smart contracts and the Ethereum transaction (should not be confused with business transactions) database. Each contract has storage options in the form of state variables (including objects and arrays of objects). Despite that, we do not store raw farm data in the contracts as it would be computationally expensive to maintain and the owners will feel a loss of control with their data. Instead, we use a separate Relational Database Management System (RDBMS) to store raw farm data. We used MySQL in our test bench, however, any SQL based DBMS can be used. In each premise running a local farm RDBMS, there are two databases (db) with identical schema: present and archive. The present db contains information about the animals currently owned/maintained by the farm. The archive database is for storing historical data of the animals which existed in the farm once but were sold. Once an animal is sent out to another farm, all its relevant data from the present database is moved to the archive database. The new owners of the animal may request data during the purchase or sometime in the future. However, the decision of how long the data should be kept in the archive and what data could be shared is at the farm owners (hence, data owners) discretion.

As already mentioned, both the present and the archive dbs contain the same structure of tables. The db schema is shown in Figure 14. The `animals` table is the root table. Each animal is uniquely identified by the tag. The id fields in the table can vary and are only used for indexing and linking data. The true animal id that remains unchanged throughout the animal's lifetime is the `tag` number which is also used by the contracts to identify animals (`animalId`). The entries of the `inspections`, `vaccines`, and `movements` table are linked to the `animals` table via the use of foreign key, `animal_id`. In these 4 animal data tables, we concatenate the key fields (excluding `id`, `animal_id` fields) in each row entry and compute a SHA3-224 hash of the concatenated string. This hash is stored in the last field of each row entry.

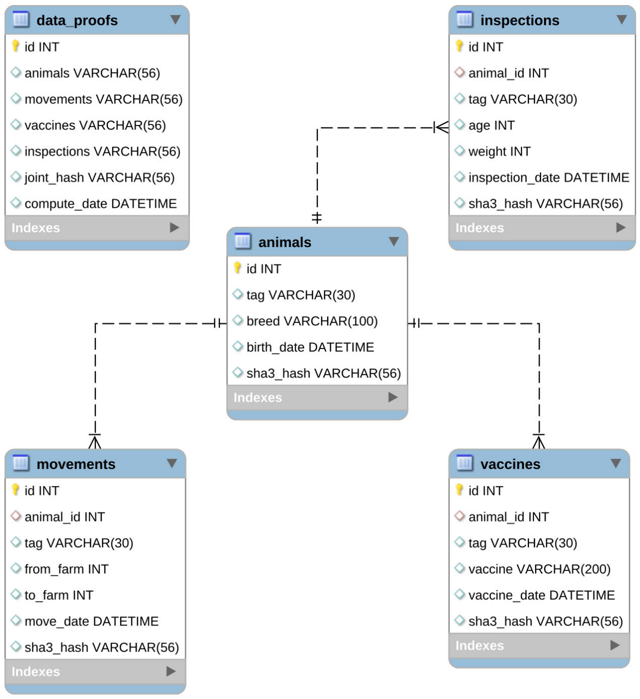


FIGURE 14. **An entity relationship diagram depicting the database tables and their relationships.**

For a specific row entry, the hash is computed once and do not change even if the animal undergoes a change. For example, a new movement of the animal results in a new row entry for that animal in the `movements` table. The old entries (if any) remain untouched. In order to store the cryptographic proofs of the data, we combine hashes and compute hash of the concatenated hashes. For each table, the hashes that are concatenated are ordered by the primary key and a SHA3-224 hash is computed for the string of hashes. The resultant hash is called a table hash. We store the table hashes in the respective fields of the `data_proofs` table. Once again, the four table hashes in a row entry of the `data_proofs` table are concatenated and a single hash is computed, which is called `joint_hash`. This is the top-level hash for a single farm database and is stored in the `FarmManager` contract of the blockchain.

APPENDIX C
TEST SYSTEM CONFIGURATION

The test simulation was run on a system whose configuration is described in Table 2. We use both parity [32] and geth [31] as the Ethereum clients as they support proof of authority (PoA) (also called 'clique' in geth) as a consensus protocol. Both parity and geth can produce similar results. The computational costs (gas cost) were computed by connecting to parity nodes while the integration tests were performed by creating a prototype running system using geth. For the prototype system, we ran 6 geth nodes in the same Linux system with separate data directory, configuration file, keystore, and port numbers for each node. For each node, we created a single user account: which

TABLE 2. **Test bench configuration.**

Processor	Core i5-3470 3.20 GHz x 4
Memory	8138 MB
Storage	R/W 550/400 MB/s
Kernel	Linux 4.15.0-96-generic
OS	Ubuntu 18.04.4 LTS
Architecture	x86-64
Parity version	2.7.2
Geth version	1.9.13
Remix version	0.10.1
Solidity compiler version	0.5.3
Consensus protocol	Proof of Authority (PoA)
Step duration	5 seconds
No. of authorities (signers)	5

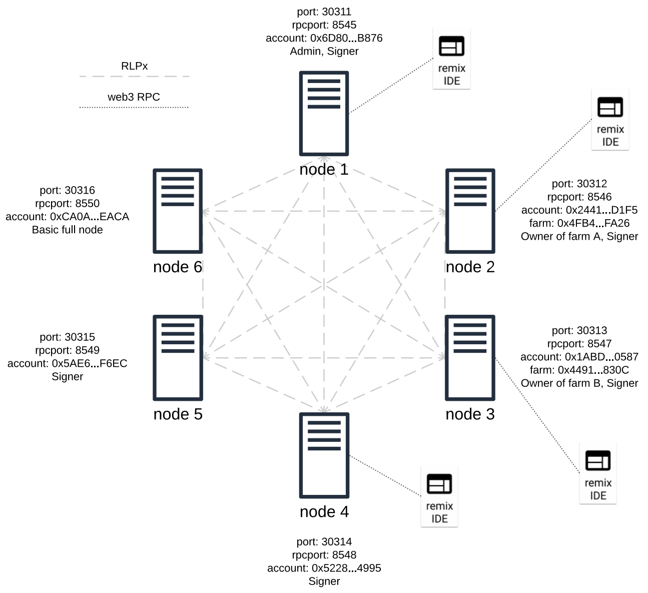


FIGURE 15. **The prototype running system which was used for integration tests. The 6 geth nodes are shown connected to each other in a P2P network. Each node is configured with unique ports for RPC and RLPx communications. Each node has an associated user account with an Ethereum address as shown above. Five out of six nodes were configured as signers, the first node was used as Admin. Remix IDE was used to communicate with the nodes.**

would either take part in PoA consensus to validate and sign blocks or deploy smart contracts and invoke contract methods. We configured 5 out of the 6 nodes to take part in the PoA consensus (mining nodes) and node 6 was a basic full node. We used the web-based remix [33] IDE (integrated development environment) to write, deploy, and test the smart contracts which are written in the Solidity [34] language. Contract deployment and method invocations were done from remix via web3 [35] by connecting to the running geth nodes of the network via RPC (remote procedure call) ports. The prototype system is illustrated in Figure 15.

ACKNOWLEDGMENT

The authors would like to thank Dr. David Amrine for his valuable insights into the cattle industry of Kansas, USA.

REFERENCES

[1] (2019). Animal Disease Traceability. [Online]. Available: https://www.aphis.usda.gov
[2] (2018). What is CattleTrace. [Online]. Available: https://www.cattletrace.org
[3] S. Nakamoto, *Bitcoin: A Peer-to-Peer Electronic Cash System*. Bitcoin.org, 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf
[4] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, Apr. 2014.
[5] E. Androulaki *et al.*, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proc. 13th EuroSys Conf.*, Apr. 2018, p. 30.
[6] Q. Yang, D. Gruenbacher, J. L. Heier Stamm, G. L. Brase, S. A. DeLoach, D. E. Amrine, and C. Scoglio, "Developing an agent-based model to simulate the beef cattle production and transportation in southwest kansas," *Phys. A, Stat. Mech. Appl.*, vol. 526, Jul. 2019, Art. no. 120856.
[7] A. Banerjee, "Integrating blockchain with ERP for a transparent supply chain," Infosys Limited, Bengaluru, India, Tech. Rep., 2018. [Online]. Available: https://www.infosys.com/oracle/white-papers/documents/integrating-blockchain-erp.pdf
[8] (2020). *TradeLens*. [Online]. Available: https://www.tradelens.com/
[9] (2019). *Provenance Tuna Tracking*. [Online]. Available: https://www.provenance.org/tracking-tuna-on-the-blockchain
[10] Q. Lu and X. Xu, "Adaptable blockchain-based systems: A case study for product traceability," *IEEE Softw.*, vol. 34, no. 6, pp. 21–27, Nov. 2017.
[11] B. Düdder and O. Ross, "Timber tracking: Reducing complexity of due diligence by using blockchain technology," Univ. Copenhagen, Copenhagen, Denmark, Tech. Rep., 2017. [Online]. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3015219
[12] S. Saberi, M. Kouhizadeh, J. Sarkis, and L. Shen, "Blockchain technology and its relationships to sustainable supply chain management," *Int. J. Prod. Res.*, vol. 57, no. 7, pp. 2117–2135, Apr. 2019.
[13] R. Casado-Vara, J. Prieto, F. D. la Prieta, and J. M. Corchado, "How blockchain improves the supply chain: Case study alimentary supply chain," *Procedia Comput. Sci.*, vol. 134, pp. 393–398, Jan. 2018.
[14] M. Montecchi, K. Plangger, and M. Etter, "It's real, trust me! Establishing supply chain provenance using blockchain," *Bus. Horizons*, vol. 62, no. 3, pp. 283–293, May 2019.
[15] K. Leng, Y. Bi, L. Jing, H.-C. Fu, and I. Van Nieuwenhuyse, "Research on agricultural supply chain system with double chain architecture based on blockchain technology," *Future Gener. Comput. Syst.*, vol. 86, pp. 641–649, Sep. 2018.
[16] H. M. Kim and M. Laskowski, "Toward an ontology-driven blockchain design for supply-chain provenance," *Intell. Syst. Accounting, Finance Manage.*, vol. 25, no. 1, pp. 18–27, Jan. 2018.
[17] F. Tian, "A supply chain traceability system for food safety based on HACCP, blockchain & Internet of Things," in *Proc. Int. Conf. Service Syst. Service Manage.*, Jun. 2017, pp. 1–6.
[18] M. Westerkamp, F. Victor, and A. Küpper, "Blockchain-based supply chain traceability: Token recipes model manufacturing processes," 2018, *arXiv:1810.09843*. [Online]. Available: http://arxiv.org/abs/1810.09843
[19] X. Liang, S. Shetty, D. Tosh, C. Kamhoua, K. Kwiat, and L. Njilla, "ProvChain: A blockchain-based data provenance architecture in cloud environment with enhanced privacy and availability," in *Proc. 17th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGRID)*, May 2017, pp. 468–477.
[20] K. Toyoda, P. T. Mathiopoulos, I. Sasase, and T. Ohtsuki, "A novel blockchain-based product ownership management system (POMS) for anti-counterfeits in the post supply chain," *IEEE Access*, vol. 5, pp. 17465–17477, 2017.
[21] H. Min, "Blockchain technology for enhancing supply chain resilience," *Bus. Horizons*, vol. 62, no. 1, pp. 35–45, Jan. 2019.
[22] Y. Yuan and F.-Y. Wang, "Towards blockchain-based intelligent transportation systems," in *Proc. IEEE 19th Int. Conf. Intell. Transp. Syst. (ITSC)*, Nov. 2016, pp. 2663–2668.
[23] R. Miller. (2018). *Walmart is Betting on the Blockchain to Improve Food Safety*. [Online]. Available: https://techcrunch.com/2018/09/24/walmart-is-betting-on-the-blockchain-to-improve-food-safety/
[24] (2019). *Wyoming Beefchain*. [Online]. Available: https://beefchain.com/
[25] S. Wang, Y. Zhang, and Y. Zhang, "A blockchain-based framework for data sharing with fine-grained access control in decentralized storage systems," *IEEE Access*, vol. 6, pp. 38437–38450, 2018.
[26] Z. Lu, W. Liu, Q. Wang, G. Qu, and Z. Liu, "A privacy-preserving trust model based on blockchain for VANETs," *IEEE Access*, vol. 6, pp. 45655–45664, 2018.
[27] D. C. Nguyen, P. N. Pathirana, M. Ding, and A. Seneviratne, "Blockchain for secure EHRs sharing of mobile cloud based E-health systems," *IEEE Access*, vol. 7, pp. 66792–66806, 2019.
[28] H. R. Hasan and K. Salah, "Blockchain-based proof of delivery of physical assets with single and multiple transporters," *IEEE Access*, vol. 6, pp. 46781–46793, 2018.
[29] K. Salah, N. Nizamuddin, R. Jayaraman, and M. Omar, "Blockchain-based soybean traceability in agricultural supply chain," *IEEE Access*, vol. 7, pp. 73295–73305, 2019.
[30] Q. Lin, H. Wang, X. Pei, and J. Wang, "Food safety traceability system based on blockchain and EPCIS," *IEEE Access*, vol. 7, pp. 20698–20707, 2019.
[31] (2020). *Go Ethereum*. [Online]. Available: https://geth.ethereum.org/
[32] (2020). *Parity Ethereum*. [Online]. Available: https://www.parity.io/ethereum/
[33] (2020). *Remix*. [Online]. Available: http://remix.ethereum.org/
[34] (2020). *Solidity*. [Online]. Available: https://solidity.readthedocs.io
[35] (2020). *Web3*. [Online]. Available: https://web3js.readthedocs.io

TANVIR FERDOUSI (Graduate Student Member, IEEE) received the B.Sc. degree in electrical and electronic engineering from the Bangladesh University of Engineering and Technology, Dhaka, Bangladesh, in 2013. He is currently a Graduate Student with the Department of Electrical and Computer Engineering, Kansas State University. Prior to joining Kansas State University, he was a Senior Software Engineer at the Samsung Research and Development Institute Bangladesh (SRBD), Dhaka, where he worked from 2013 to 2016. His research interests include computer networks, network theory, and epidemic models.

DON GRUENBACHER (Member, IEEE) received the B.Sc., M.Sc., and Ph.D. degrees from Kansas State University, Manhattan, KS, USA, in 1989, 1991, and 1994, respectively, all in electrical engineering. He is currently serving as the Department Head and an Associate Professor with the Department of Electrical and Computer Engineering, Kansas State University. Prior to joining Kansas State University as a Faculty Member, he was a Member of Senior Staff with the Space Department, Johns Hopkins University Applied Physics Laboratory, Laurel, MD, USA, from 1994 to 1997, and from 1989 to 1990. His research interests include the areas of computer networks, communications, and digital design. He has been recognized as an Outstanding Faculty Member by the Eta Kappa Nu and the Mortar Board.

CATERINA M. SCOGLIO (Senior Member, IEEE) received the Dr.Eng. degree from the Sapienza University of Rome, Italy, in 1987. She is currently the Paslay Chair Professor with the Department of Electrical and Computer Engineering, Kansas State University. Before joining Kansas State University, she worked at the Fondazione Ugo Bordoni, from 1987 to 2000, and the Georgia Institute of Technology, from 2000 to 2005. She is also affiliated with the Institute of Computational Comparative Medicine (ICCM), Kansas State University, as a Faculty Member. Her main research interests include network science and engineering, and modeling and analysis of complex networks, with applications in epidemic spreading and power grids.