applied sciences

MDPI

Article

The Design and Implementation of a Secure Datastore Based on Ethereum Smart Contract

Izdehar M. Aldyaflah [1], Wenbing Zhao [1,*], Himanshu Upadhyay [2] and Leonel Lagos [2]

[1] Department of Electrical Engineering and Computer Science, Cleveland State University, Cleveland, OH 44115, USA; aldyaflahizdehar@yahoo.com
[2] Applied Research Center, Florida International University, Miami, FL 33174, USA; upadhyay@fiu.edu (H.U.); lagosl@fiu.edu (L.L.)
* Correspondence: wenbing@ieee.org

Abstract: In this paper, we present a secure datastore based on an Ethereum smart contract. Our research is guided by three research questions. First, we will explore to what extend a smart-contract-based datastore should resemble a traditional database system. Second, we will investigate how to store the data in a smart-contract-based datastore for maximum flexibility while minimizing the gas consumption. Third, we seek answers regarding whether or not a smart-contract-based datastore should incorporate complex processing such as data encryption and data analytic algorithms. The proposed smart-contract-based datastore aims to strike a good balance between several constraints: (1) smart contracts are publicly visible, which may create a confidentiality concern for the data stored in the datastore; (2) unlike traditional database systems, the Ethereum smart contract programming language (i.e., Solidity) offers very limited data structures for data management; (3) all operations that mutate the blockchain state would incur financial costs and the developers for smart contracts must make sure sufficient gas is provisioned for every smart contract call, and ideally, the gas consumption should be minimized. Our investigation shows that although it is essential for a smart-contract-based datastore to offer some basic data query functionality, it is impractical to offer query flexibility that resembles that of a traditional database system. Furthermore, we propose that data should be structured as tag-value pairs, where the tag serves as a non-unique key that describes the nature of the value. We also conclude that complex processing should not be allowed in the smart contract due to the financial burden and security concerns. The tag-based secure datastore designed this way also defines its applicative perimeter, i.e., only applications that align with our strategy would find the proposed datastore a good fit. Those that would rather incur higher financial cost for more data query flexibility and/or less user burden on data pre- and post-processing would find the proposed database too restrictive.

Keywords: blockchain; smart contract; data immutability; datastore; user access control; role-based authentication; gas consumption; IPFS

Citation: Aldyaflah, I.M.; Zhao, W.; Upadhyay, H.; Lagos, L. The Design and Implementation of a Secure Datastore Based on Ethereum Smart Contract. *Appl. Sci.* **2023**, *13*, 5282. https://doi.org/10.3390/app13095282

Academic Editor: Leandros Maglaras

Received: 24 March 2023
Revised: 20 April 2023
Accepted: 21 April 2023
Published: 23 April 2023

1. Introduction

Data security is of paramount importance for all systems [1,2]. Traditionally, data are stored in relational database systems or NoSQL datastores [3]. Unfortunately, data stored this way are not only vulnerable to theft, but could easily be modified as well [4,5]. While data theft clearly is a security breach, compromised data could lead to more serious integrity problems [6]. Blockchain technology [7], on the other hand, offers a new way of storing data with the immutability property if the blockchain system has sufficient scale [8].

There are primarily two ways of storing data with the blockchain [7]. One way is to store data directly as part of a transaction, which is possible for most public blockchain systems. The other way is to store data via a smart contract. Directly storing data as part of a transaction costs much less and it is also much less complex than using a smart

contract [9]. However, directly storing data as part of a blockchain transaction suffers from several limitations: (1) there is no access control over who may write the data to the blockchain and who may read the data from the blockchain; (2) there is no support for querying the data stored in the blockchain systematically, i.e., one would have to know the transaction hash to locate the particular transaction; (3) the amount of data can be packed into a transaction is very limited due to the predefined transaction structure.

While using a smart contract may address all three issues, there is a lack of universally accepted guidelines on designing secure datastore based on smart contracts. The current study is an attempt towards this goal and is driven by three research questions. First, we will explore to what extend a smart-contract-based datastore should resemble a traditional database system. Second, we will investigate how to store the data in a smart-contract-based datastore for maximum flexibility while minimizing the gas consumption. Third, we seek answers regarding whether or not a smart-contract-based datastore should incorporate complex processing such as data encryption and data analytic algorithms. The proposed smart-contract-based datastore aims to strike a good balance between several constraints: (1) smart contracts are publicly visible, which may create a confidentiality concern for the data stored in the datastore; (2) unlike traditional database systems, the Ethereum smart contract programming language (i.e., Solidity) offers very limited data structures for data management; (3) all operations that mutate the blockchain state would incur financial costs and the developers for smart contracts must make sure sufficient gas is provisioned for every smart contract call, and ideally, the gas consumption should be minimized.

Our investigation shows that although it is essential for a smart-contract-based datastore to offer some basic data query functionality, it is impractical to offer query flexibility that resembles that of a traditional database system. Defining a complex schema for highly structured data as commonly used in traditional database systems could compromise the confidentiality because the smart contract is publicly visible. Adopting a more complex structure for data storage in a smart contract would significantly increase the financial cost (in terms of gas consumption) for insertion operations. Furthermore, Solidity lacks built-in support for some database system features, such as the auto increment primary key. Hence, it is both desirable and necessary to adopt basic structures for data storage and for data query.

Furthermore, we propose that data should be structured as tag-value pairs, where the tag serves as a non-unique key that describes the nature of the value. This idea is inspired by the pervasive use of key-value pairs in modern NoSQL datastore and in Ethereum. We intentionally allow the tag to be non-unique so that multiple data entries may share the same tag. This design offers a high degree of flexibility for applications. For example, in sensing data logging, it may be desirable to use the same tag for all data generated by the same sensor in the same day [10]. We note that this design allows the users to insert tag-value pairs with unique tags.

We also conclude that complex processing should not be allowed in a smart contract due to the financial burden and security concerns [11,12]. While adding complex processing could improve the usability of the datastore because the users are relieved from having to perform such processing, the complexity of the smart contract could be drastically increased. Higher complexity could sharply increase the gas consumption and increase the likelihood of introducing vulnerabilities to the smart contract [11,12].

The tag-based secure datastore designed this way also defines its applicative perimeter, i.e., only applications that align with the chosen answers to the research questions would find the proposed datastore a good fit. Those who would rather incur higher financial cost for better data query flexibility and/or less user burden on data pre- and post-processing would find the proposed database too restrictive. We follow a design principle that if an operation can be conducted off-chain (i.e., outside the smart contract), then the operation should be moved out of the smart contract to keep the smart contract as simple as possible. A simple smart contract would not only reduce gas consumption, but also minimize the likelihood of introducing security vulnerabilities. Another apparent applicative perimeter

is that the proposed datastore is based on Ethereum. Users who wish to use other public blockchains would have to port our design to the desirable blockchain.

We argue that the tag-based datastore provides the best tradeoff for query flexibility, data confidentiality, smart contract security, and minimizing the financial cost of the datastore. The first benefit of this design is that it helps protect the confidentiality of the data stored in the datastore. By encoding the tag and the data as strings, it is difficult for an adversary to understand what the original data are. The encoding method can be as simple as mapping the type of sensor (e.g., temperature) to an integer and can be as complex as encrypting the data using AES. The datastore itself does not care how the tag and the data entry are encoded, i.e., the datastore treats the tag and data as opaque objects and only offers storage and query functionalities. The user of the datastore gets to decide the level of protection of the tag and the data entry. Indeed, this aligns with the guideline for designing smart contracts [11].

The second benefit of this design is simplicity. By treating the tag and the data as opaque objects, the insertion and query operations become simple and much less error-prone.

The third benefit of this design is flexibility. In addition to the flexibility for achieving data confidentiality as we mentioned earlier, this design also offers the flexibility of system integration. For applications that only have a small amount of data to log with a low data generation rate, the data can be directly written to the datastore after proper encoding. That said, we caution that even with a moderate data generation rate, storing all data in the datastore via a smart contract will not only incur excessive financial cost, but also may exceed the throughput of the blockchain system. To address this concern, we propose to only store the digest of the data entry in the smart contract and log the full data entry in other peer-to-peer file systems, most notably, the InterPLanetary File System (IPFS) (https://docs.ipfs.tech/concepts/what-is-ipfs/ (accessed on 1 March 2023)). This would minimize the financial cost and significantly lower the throughput demand on the blockchain system. If the generation rate is high, it may be necessary to first aggregate the data and record the hash of the aggregated data in the blockchain, as we have done previously [13].

In summary, this paper makes the following research contributions:

- In this study, we raise three research questions and present our answers by considering the differences between traditional database systems and Ethereum smart contract. Driven by these research questions, we aim to propose a set of guidelines on datastore design based on Ethereum smart contract;
- We introduce a smart contract that functions as the proposed secure datastore. The datastore allows role-based access control and tag-based query of data in the datastore while protecting the data confidentiality. The design of the smart contract is intentionally simple yet flexible that accommodates different data generation rates, different levels of data confidentiality requirements, and different levels of financial budget.
- We fully characterize the financial cost (i.e., gas consumption) of the datastore in terms of gas consumption under a variety of scenarios and compare with competing solutions;
- We provide the design, implementation, and evaluation of a system that integrates the proposed datastore and IPFS, where the smart contract is used to store the digest of each data entry, and the IPFS is used to store the full data entry. We demonstrate the advantages of this integrated solution for secure data storage and retrieval in terms of gas consumption and operation latency.

The remainder of the paper is organized as follows. Section 2 presents the related work and highlights our research contributions. Section 3 elaborates the details of our smart contract design. Section 4 discusses the system integration issues with the proposed smart contract for secure data storage. Section 5 reports the experimental results. Section 6 concludes the paper and outlines future research directions.

2. Related Work

This study is about relying on blockchain to securely store data. The related studies reported in the literature can be roughly divided into three categories: (1) custom-built

blockchain systems designed specifically for secure data storage (such as [14,15]); (2) storing data directly in transactions using public blockchains; and (3) storing data in smart contract using public blockchains.

Although the work in the first category has some merit, for example, a system was designed to handle large-scale Internet of Things (IoT) data with data trading functionalities in [14], and another system was designed to self-repair corrupted data with local repair groups in [15], the proposed systems are not actually implemented, let alone available for other researchers to use for secure data storage. As such, we do not draw comparisons with this line of work.

Studies in the second category predominately used the IOTA distributed ledger IOTA [16,17]. IOTA was designed to support high-throughput transactions using a direct acyclic graph instead of a single chain of blocks. Furthermore, IOTA does not impose a transaction fee. To mitigate spamming attacks, all clients must perform a moderate amount of proof-of-work in every transaction submitted to the IOTA network. IOTA provides application programming interfaces (APIs) for the construction of a transaction that only contains data. The transaction size in IOTA is limited to 32KB. In 2017, IOTA introduced the masked authenticated message (MAM) to facilitate secure communication between IoT devices and IOTA (https://github.com/iotaledger/mam.client.js/ (accessed on 1 March 2023)). A number of studies used IOTA MAM for securing IoT data [18–20], for supply chain applications [21], for health data sharing [22,23] and management [24,25], for manufacturing applications [26,27], for securing industry control [28], for securing system logs [29], for securing vehicular networks [30,31], and for securing power systems [32].

Previously, we also focused on using IOTA to store sensing data securely [10]. The system adopted a hierarchical aggregation scheme for the sensing data [33] to reduce the amount of data to be placed on IOTA. Recently, we extended the work to store sensing data directly in the data field of Ethereum transactions [13]. Our previous work has a number of limitations as a result of storing of data directly as part of blockchain transactions. A major concern of the approach is the lack of user access control regarding who may upload the data into the blockchain. Not addressing this issue could lead to serious security issues because the integrity of the data may be compromised if adversaries could upload faulty data to the blockchain. Second, the data stored in the blockchain are publicly accessible, which could present confidentiality and privacy issues. By adding user access control, we address both concerns. Third, it is difficult to query the data stored in the blockchain. One would have to know the transaction hash to retrieve the data stored in a particular transaction. In [13], we proposed a workaround by introducing an indexing scheme. However, the indexing data must also be stored securely. If the indexing data are compromised or lost, data query would not be possible. In the current study, the datastore offers the functionality of data queries.

In the remainder of this section, we consider related studies in the third category. We note that although there is a large body of work on smart contract applications and this line of work has been well reviewed (for example, [34,35]), most such studies focus on using smart contracts for access control and coordination. A small fraction of studies are dedicated to secure data storage using smart contracts. An even smaller fraction of studies provided details on smart contract operations for secure data storage and retrieval [36–39], which are the only works directly related to our current study. We discuss these studies by looking at how they responded to the three research questions.

2.1. Query Flexibility of Smart-Contract-Based Datastores

Most studies attempted to develop smart contracts for data storage in a way similar to traditional database systems from two perspectives: (1) structured data are explicitly specified in the smart contract; and (2) users may query the data in the smart contract using a filter for a result directly needed by the users. These studies are predominately in the biomedical and healthcare domain for medical data storage and query [36–38,40], and most are winning entries in the 2019 secure genome analysis competition [36,37].

2.2. Data Structures Used in Smart-Contract-Based Datastores

Ethereum offers limited data structures for data storage and indexing. Data entries may be added into an array of predefined types (such as string or bytes32) or custom types (such as a complex structure with multiple fields). The latter would be quite similar to tables in traditional database systems. However, unlike tables in traditional database systems, there is no need to specific primary keys and the developer must manually keep track of the index to each entry. The only form of indexing in Ethereum is mapping, which is a hash table that maps a key to one or more values. To facilitate data query, one mapping is typically needed for each required filter criterion.

In [36], three smart contract solutions that were developed for the 2019 secure genome analysis competition were presented. The three solutions won the first place, the second place, and the honorable mention. The first-place solution is referred to as query index. The second-place solution is referred to as index everything. The honorable mention solution is referred to as dual-scenario indexing.

In query index [36], a single mapping and an array are used to store the gene-variant-drug observations, as shown in Figure 1. Each of the gene-variant-drug observations is appended to the array (Step 1 in Figure 1). The value of the mapping is the list of indices to the observation array. To facilitate query using either gene-variant-drug or a combination of gene/variant/drug with one or more wildcards, for each insertion of a unique gene-variant-drug observation, eight keys (i.e., gene-variant-drug, gene-variant-*, *-variant-drug, gene-*-drug, gene-*-*, *-variant-*, *-*-drug, *-*-*) are inserted into the mapping, and the index of the observation to the array is appended to the list of indices stored as the value of the mapping (Step 2 in Figure 1). Here * denotes a wildcard, meaning that the user wishes to query the data that contain all possible values for this field.

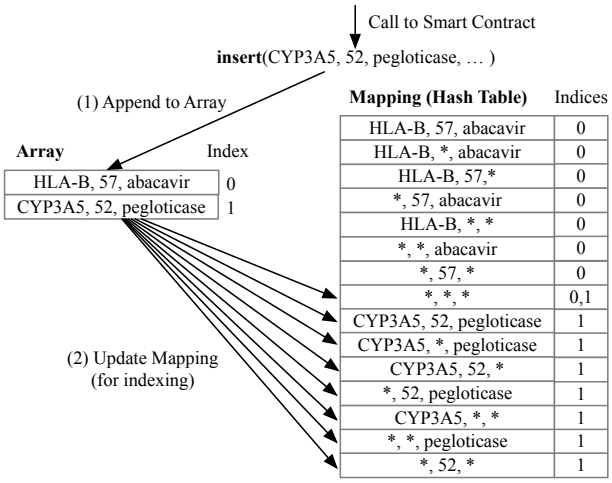

Figure 1. Data structures used in the query index scheme and how they are used in an insert call to the smart contract.

Unlike the first-place solution, index everything assumes that the number of gene, variant, and drug is small enough to be represented as an 8-bit integer [36]. Based on this assumption, a 24-bit unsigned integer can be used to encode all gene–variant–drug combinations. The counts for each of the possible outcomes for each gene-variant-drug observation are recorded as the value to the mapping. To capture all possible outcomes, five mappings are used for the five possible outcomes: improved, unchanged, deteriorated, suspected relation, and side effect. As shown in Figure 2, when the insert function is invoked, the key is first constructed in Step 1, and then the five mappings are updated in Step 2. In the figure, the 8-bit integer and the 24-bit integer are represented as a hexadecimal numbers. All hexadecimal numbers are preceded with $0x$.

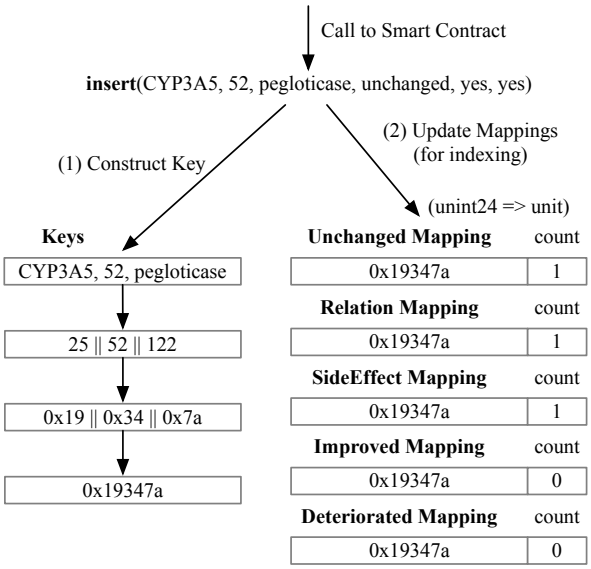

Figure 2. Data structures used in the index everything scheme and how they are used in an insert call to the smart contract.

In the dual-scenario indexing solution [36], two mappings (GeneDrugRelationKeyMapping and GeneData), one structure called GeneDrugRelation, and an GeneDrugRelation array are used to store the observation entries and to facilitate queries with the full gene-variant-drug and with wild cards, as shown in Figure 3. The GeneDrugRelation structure contains the gene name, variant number, drug name, and various outcome counts and percent. The GeneData mapping maps the gene–variant–drug combination to a GeneDrugRelation array. The GeneDrugRelationKeyMapping mapping maps all possible combinations with wild cards to gene–variant–drug combinations. The latter mapping is pre-populated and is used entirely to facilitate queries with one or more wildcards.

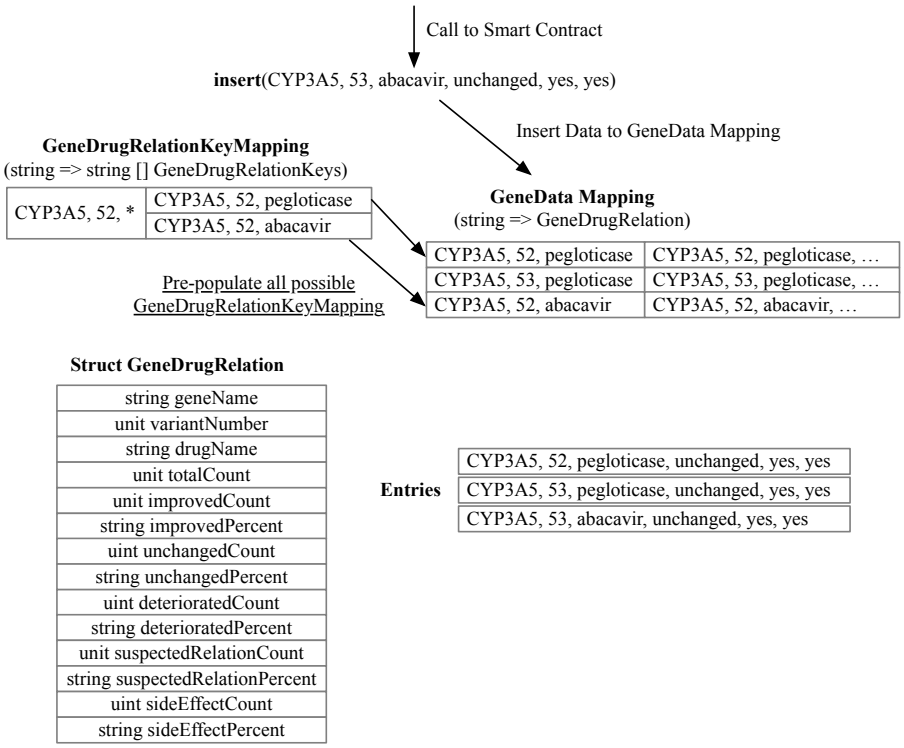

Figure 3. Data structures used in the dual scenario scheme and how they are used in an insert call to the smart contract.

In [37], two smart contracts that won the third place in the same competition are presented in great details. We refer to this solution as full indexing. Both smart contracts in [37] offer identical insert and query interfaces to users. The data structures used in this solution consist of one array (called Database) to store the observation entries, and three mappings for gene, variant, and drug, respectively. As shown in Figure 4, when the insert function is invoked, the data entry is first appended to the Database array in Step 1, and then the three mappings are updated in Step 2.

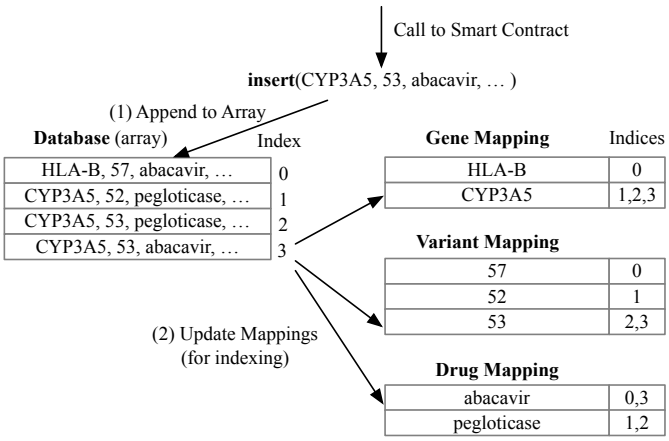

Figure 4. Data structures used in the full indexing scheme and how they are used in an insert call to the smart contract.

In [38], an Ethereum smart contract was used to store and query fictitious COVID-19 patient data. The design of the smart contract is rather similar to the first smart contract introduced in [37]. Each observation consists of the COVID-19 variant (of string type), patient ID (of unsigned integer type), comorbidity (of string type), chest CT severity grade (of unsigned integer type), and patient age (of unsigned integer type). The first three parameters are used together as the key to the entry.

All these studies focused on quantifying the insertion and query latency, and space utilization of the data stored in the smart contract. User access control is notably missing in these studies. While there is some discussion regarding minimizing the gas usage in the store and query parameters in [37], the overall financial cost for the datastore was not investigated.

The only study that is not in the field of healthcare is [39], where one array is used to store IoT data. The focus of this study is to characterize the gas consumption in two alternative schemes. The first design assumes that the contract simply appends all IoT data into an array (referred to as array appending). The second design assumes that the contract uses a fixed array of size N to store the IoT data where only the latest N data entries are kept (i.e., old data entries are overwritten by the newly arrived data entries once the array is full). The second design is referred to as array substitution. In both designs, the gas cost grows linearly with the number of IoT data entries logged in the contract. The substitution cost per data entry is 39,305 gas, which is slightly cheaper than that for recording a new data entry (i.e., 52,960 gas). The paper did not consider user authentication, user management, and data query issues.

Before ending this subsection, we comment on how data should be stored in smart contracts. The existing solutions all use a single array to store the data entries. To facilitate data query, either multiple mappings are used or extra entries are created in a single mapping. We argue that a single mapping with one tag as the key is sufficient for practical use with less gas consumption, provided that the users are willing to encode the tag properly and extract data from the result of the single-tag query, as we will show in later sections.

2.3. Complex Processing and Smart Contract

It appears that it is common practice to avoid incorporating complex processing in smart contracts. This is quite different from traditional database systems, which offer various stored procedures for automatic data analytics. Studies that wish to protect the confidentiality/privacy of the data always encrypt the data off-chain and the smart contract is often used for key management and user access control [41,42]. In [41], re-encryption is used off-chain to secure the IoT data. In [42], searchable encryption is used, again off-chain, to ensure the security of the sharing of electronic health records.

Although it is clear that encryption of data should be performed off-chain, the data structures used in the smart contract must be compatible with storing encrypted data. The tag-based data structures in our smart contract are designed to be compatible with storing encrypted tags and data values.

2.4. Novel Contributions of Our Study

The most prominent novel contribution of our study is the recognition of the fundamental differences between smart contracts and traditional database systems. The current study is guided by the three research questions we elaborated above. The answer to the first research question has led us to choose a strategy of providing a basic tag-based data query function in the smart contract. The objective is to keep the smart contract simple so that there is little chance of introducing vulnerabilities and the gas consumption is minimized. The answer to the second research question has led to us using only a single string array and a single mapping that maps a given tag (of the string type) to a value (also of the string type). As we will show in later sections of this paper, we demonstrate that this design minimizes the gas consumption and has sufficient expressibility and flexibility. The data structure design in our approach is also influenced by the answer to the third research question. Although it is commonplace to avoid complex processing in the smart contract, the design of the smart contract must be conducive to meeting user requirements such as data confidentiality and privacy. In our design, both the tag and the data value are treated as opaque strings where the user could use encrypted tags and data values. In contrast, the existing solutions in [36,37] would have to make significant changes if the genome data must be encrypted.

3. Smart Contract Design

In this section, we present the details of our smart contract design. As we report earlier in Section 2, academic studies on using smart contract for data storage and query are typically focused the functionality and the storage and query efficiency in terms of latency and space utilization. In our study, what prompted us to consider smart contract is to protect the security of the data placed on the blockchain as part of a secure sensing data processing and logging system [10]. Without proper access control, public functions defined in a smart contract may be invoked by any user who knows the contract address. For sensing data and many other forms of data, such as biomedical, healthcare, and electronic medical records, it is essential to allow only legitimate users to upload data into the smart-contract datastore because otherwise the integrity of the data cannot be guaranteed. For confidentiality of the data, only authorized users may query the data from the smart contract. In this paper, we show how to add user access control as part of the smart contract for data storage and query. The smart contract is written in Solidity, the official programming language for Ethereum smart control. The full smart contract source code is provided in Appendix A.

3.1. User Access Control and User Management

The user access control algorithm in the smart contract is described in Algorithm 1. We define three roles: Administrator (Admin for short), Writer, and Reader. The Admin has all the privileges, including user management, the right to write data into the smart contract, and the right to read from the smart contract. The Writer has the right to write data into the smart contract and the right to read from the smart contract, but does not have user

management privilege. The Reader only has the right to read from the smart contract. This design means that for a user (i.e., the corresponding account) to read from or write to the smart contract, the user must be added to the list of accounts with proper roles. Each role is represented by a 32-byte long byte array (using the Solidity bytes32 type) obtained by hashing a string literal ("Admin" for the Admin, "Write" for the Writer, and "Read" for the Reader) using the `keccak256` secure hash function (A represents the Admin, W represents the Writer, and R represents the Reader).

Access control is enforced for all public functions defined in the smart contract using a modifier called ONLYROLE. As shown in lines 1–12 in Algorithm 1, this modifier takes one argument as the minimum role for the called operation, and checks if the calling account has already been assigned a role that is the same or more privileged. The call is rejected if the check fails.

By default, the account that created the smart contract assumes the Admin role. This account may add other accounts as Admin, Writer, or Reader via the GRANTADMINROLE, GRANTWRITERROLE, and GRANTREADERROLE functions. To keep track of the accounts that have been authorized with Admin, Writer, or Reader roles, a mapping `roles` is defined, as shown in Figure 5. In Solidity, a mapping is a hash table that takes a key and a value. The `roles` mapping would map a role into another mapping that maps an account address to a boolean value. In cases where multiple accounts have Admin, Writer, or Reader role, the value of the `roles` mapping would resemble an array. However, unlike an array, the value in a mapping cannot be iterated. One would have to know the account address in the value of the `roles` mapping to know whether the account has been granted one of the roles. The `roles` mapping is essentially a two-dimensional hash table with the role as the first key and the address of an account as the second key. If an account with an address `acct` is authorized with a particular role, then `roles[role][acct]` is set to true, as illustrated in Figure 5. It is possible that an account was first granted a particular role, but was later removed from the role, in which case, the account is still kept in value of the `roles` mapping with the Boolean value set to false.

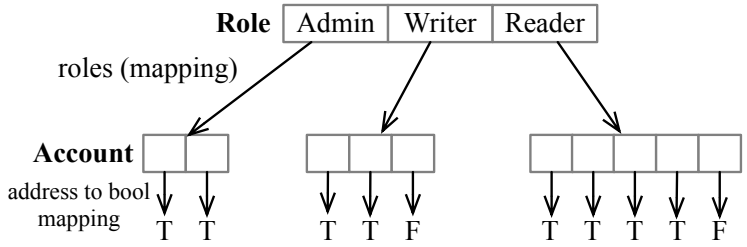

Figure 5. Data structures used for user management and their relationship.

As shown in Algorithm 1, if an account `acct` is granted the Admin role, `roles[A][acct]`, `roles[W][acct]`, `roles[R][acct]` are all set to true in lines 5–7; if an account `acct` is granted the Writer role, `roles[W][acct]`, `roles[R][acct]` are set to true in lines 10–11; and if an account is granted the Reader role, then only `roles[R][acct]` is set to true in line 14.

An Admin account may also remove the Writer or Reader role from other accounts by calling the DELETEWRITER and DELETEREADER functions. When removing a role from an account, the `roles` mapping for the corresponding account is set to false (in lines 24–25 when removing the writer role, and in line 28 when removing the reader role). Only the contract owner may remove the Admin role from an account (in line 17), and `roles[A][acct]`, `roles[W][acct]`, `roles[R][acct]` are all set to false in lines 18–20.

Algorithm 1 Algorithm for user access control.

```
Require: address owners
Require: mapping(bytes32 => mapping(address => bool) roles
Require: bytes32 A ← keccak256("Admin")
Require: bytes32 W ← keccak256("Write")
Require: bytes32 R ← keccak256("Read")
procedure ONLYROLE(role)
    require(roles[role][msg.sender], "not authorized")
end procedure
procedure GRANTADMIN(acct) ONLYROLE(A)
    roles[A][acct] ← true
    roles[W][acct] ← true
    roles[R][acct] ← true
end procedure
procedure GRANTWRITER(acct) ONLYROLE(A)
    roles[W][acct] ← true
    roles[R][acct] ← true
end procedure
procedure GRANTREADER(acct) ONLYROLE(A)
    roles[R][acct] ← true
end procedure
procedure DELETEADMIN(acct) ONLYROLE(A)
    if is owner of contract then
        roles[A][acct] ← false
        roles[W][acct] ← false
        roles[R][acct] ← false
    end if
end procedure
procedure DELETEWRITER(acct) ONLYROLE(A)
    roles[W][acct] ← false
    roles[R][acct] ← false
end procedure
procedure DELETEREADER(acct) ONLYROLE(A)
    roles[R][acct] ← false
end procedure
```

Figure 6 illustrates the four roles defined in our smart contract and related operations. The Owner is the account that deploys the smart contract, which has the highest privilege. The Owner assumes the Admin role automatically. Both the Owner and the Admin may grant other accounts the Admin, Writer, or Reader roles. However, only the Owner may delete an Admin account.

In the sensing data logging application scenario (more about this scenario in Section 4.1), the sensing data processing component would function as the Owner role. This component would also take the Admin and Writer roles because it is in charge of uploading sensing data to the datastore. If a data visualization component is present in the system, then this component would need to be granted a Reader role so that it may query the datastore.

In the secure genome analysis application scenario (more about this scenario in Section 4.2), each institution that enters the competition would function in the Owner and the Admin role because the institution would develop its own smart contract for data insertion and query as specified by the competition organizer. In a more realistic scenario where researchers could upload their genome experiment observations to the datastore for data sharing, such researchers would need to acquire the Writer role. Other researchers who are interested in downloading the genome experiment observations would need to acquire the Reader role. The institution that makes the datastore available to the researchers would function as the Owner and Admin.

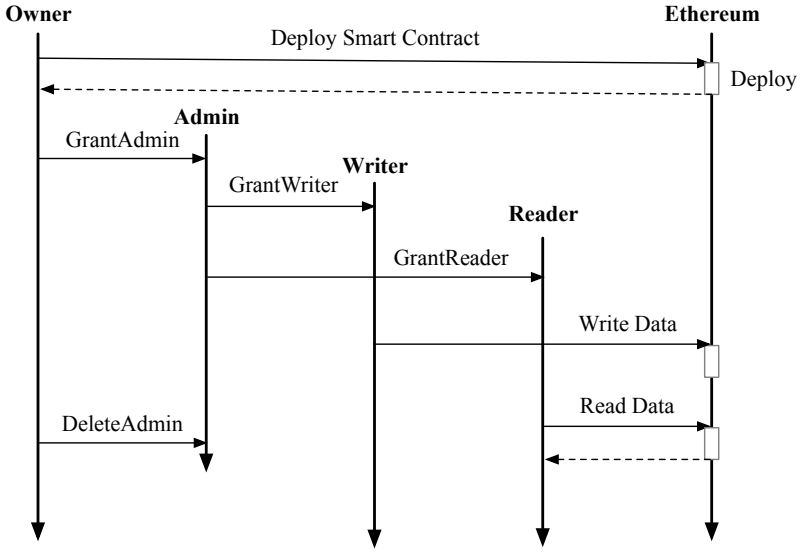

Figure 6. Different roles in the user management and user access control and their operations.

3.2. Data Storage and Retrieval

The pseudo code for the data storage and retrieval is shown in Algorithm 2. The smart contract maintains a string array called `database`, a mapping called `tagindex`, and a counter that remembers the last index to the `database`, as shown in Figure 7. The tag and the data for each write operation are regarded as opaque objects. How the tag and data are encoded and the relationship between the tag and the data to be written to the contract are completely up to the user. Although we recommend the tag to be limited to 32 bytes to save on gas, we do not impose any limit on the tag length. Furthermore, the tag can be unique among all the writes or it could be the same for multiple writes. To facilitate query of the data written to the smart contract, we offer three functions: (1) fetch all data that carry the same given tag; (2) fetch all data written to the contract; and (3) fetch a given number of most recent data entries written to the contract. If the tag and/or the data are encoded properly, one may conduct various queries on the fetched data.

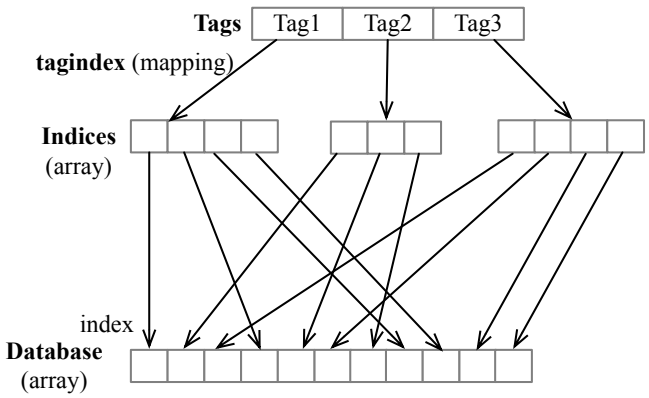

Figure 7. Data structures used for data storing and query operations.

To write data to the contract, one would call on the WRITE function with a tag and a data entry (lines 1–12 in Algorithm 2). The data entry is first appended to the database array (line 2 in Algorithm 2), then the tag is added to the `tagindex` mapping with the current `counter` as the value (line 3 in Algorithm 2). Finally, the `counter` is incremented by 1 (line 4 in Algorithm 2).

To fetch data from the contract, one would call the READ function with a tag (lines 6–12 in Algorithm 2). The indices array is retrieved from the `tagindex` mapping (line 7 in Algorithm 2). Then, the entries from the database are retrieved and added into a local result variable in a loop (lines 8–12 in Algorithm 2). Finally, the `result` is returned to the caller (line 16 in Algorithm 2).

To fetch all the data written to the contract, one would call the READALL function (lines 18–12 in Algorithm 2), where the `database` is returned to the caller (line 19 in Algorithm 2). The number of entries in the `database` can be queried via the GETDBSIZE function (lines 21–12 in Algorithm 2). One may fetch the most recent n entries from the contract via the READRECENT function. If n is the same or larger than the current size of the `database`, then the entire `database` is returned to the caller (lines 25–12 in Algorithm 2). Otherwise, the most recent n entries are retrieved, coped to a local array, and returned to the caller (lines 28–12 in Algorithm 2).

Algorithm 2 Algorithm for data write and read.

```
Require: mapping(string => uint []) tagindex
Require: string [] database
Require: uint counter
 1: procedure WRITE(tag, entry) ONLYROLE(W)
 2:     database.push(entry)
 3:     tagindex[tag].push(counter)
 4:     counter++
 5: end procedure
 6: procedure READ(tag) ONLYROLE(R)
 7:     uint [] indices ← tagindex[tag]
 8:     uint length ← indices.length
 9:     string [] result ← new string(length)
10:     i ← 0
11:     while i < length do
12:         entry ← database[indices[i]]
13:         result[i] ← entry
14:         i + +
15:     end while
16:     return result
17: end procedure
18: procedure READALL( ) ONLYROLE(R)
19:     return database
20: end procedure
21: procedure GETDBSIZE( ) ONLYROLE(R)
22:     return counter
23: end procedure
24: procedure READRECENT(n) ONLYROLE(R)
25:     if  n >= counter then
26:         return database
27:     end if
28:     string [] result ← new string[](n)
29:     i ← 0
30:     while i < n do
31:         entry ← database[counter-i-1]
32:         result[i] ← entry
33:         i + +
34:     end while
35:     return result
36: end procedure
```

4. System Integration with Smart Contract

In this section, we show how to use the proposed smart contract in a system for secure data storage and query as illustrated in Figure 8. The main components in the system consists of data producers (such as IoT devices and wireless sensors), data consumers (i.e., those who make decisions based on the data collected), a datastore frontend, IPFS, and Ethereum with the proposed smart contract.

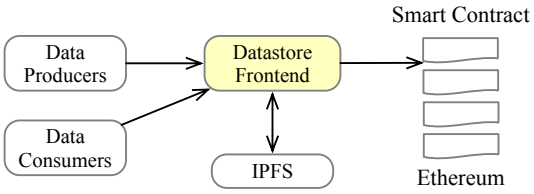

Figure 8. Main components in a system for secure data storage and query.

We assume that strong authentication is in place in the system to ensure that data producers and data consumers are authorized to store and query data via the datastore frontend. How to ensure such strong authentication is out of the scope of this study and we have reported our solution to this issue in a separate article [13]. The purpose of the datastore frontend is to provide easy-to-use APIs for data insertion and query so that data producers and data consumers do not have to handle the integration of IPFS and Ethereum smart contracts. Another benefit of using the frontend is that the system can be adapted to use with other blockchain systems by modifying the frontend component only. The smart contract design allows only authorized users to upload and download data from the smart contract.

The data producers are the entities that would like to upload data to the system. For example, in a sensing system, a sensor or IoT device could function as a data producer. In the case of the genome analysis competition, the organization that would like to enter the competition would function as a data producer. The data consumers are the entities that would like to retrieval data from the system. In a sensing system, managers and engineers could be data consumers because they would like to inspect the data to gauge if the plant being monitored is functioning normally. The IPFS is used to store the data uploaded by the data producers. The smart contract is used to store the IPFS hash generated by the IPFS. The smart contract is also in charge of performing user access control.

The interaction flow of the users (i.e., data producers and data consumers), the datastore frontend, IPFS, and the smart contract deployed on Ethereum are illustrated in Figure 9. The proposed smart contract is deployed by the datastore frontend at the begining of the system operation. To upload data, a data producer would call the `store(tag,data)` function provided by the datastore frontend. Upon being invoked, the datastore frontend would add the data to IPFS by calling `add_json(data)`. The call returns an IPFS `hash`, which will be written to the smart contract as the data entry with the given tag in the call `write(tag,hash)`. To retrieve data from the system, a data consumer would call the datastore frontend with `fetch(tag)`. Then, the datastore frontend would first issue a `read(tag)` call to the smart contract. Once the data entry is returned, which would be the IPFS `hash`, the datastore frontend issues a `get_json(hash)` call to IPFS, which would return the original data stored at IPFS. In the last step, the datastore frontend returns the data corresponding to the particular `tag`. We note that in this design, the datastore frontend would be the only entity that interacts with the smart contract, i.e., the datastore is the Owner as well as the Admin.

In the following, we first consider the scenario typical in sensor-based systems where multiple sensors would generate a large volume of sensing data. Then, we demonstrate how to use our smart contract as a potential solution for the secure genome analysis competition. Finally, we present an analysis of the security properties in the proposed secure datastore.

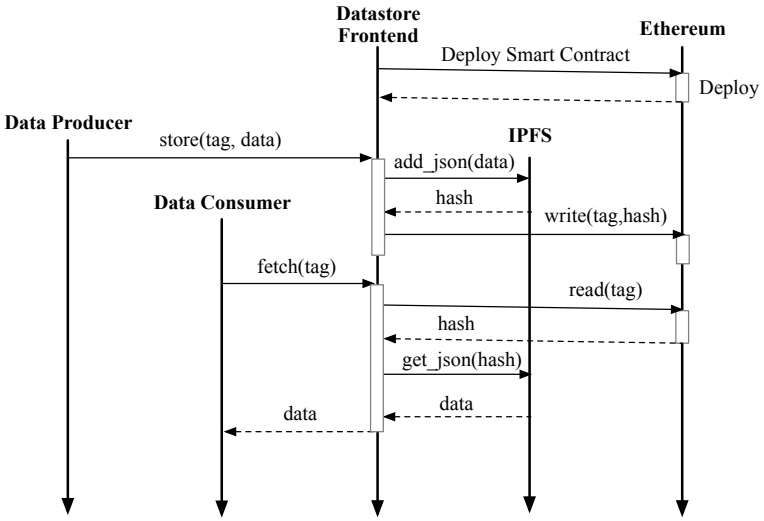

Figure 9. The interaction flow involving the user (i.e., data producers and data consumers), the datastore frontend, IPFS, and the smart contract deployed on Ethereum.

4.1. Sensing Data Logging

Sensors typically generate a large volume of sensing data with high sampling rates, as such, it is not practical to create one transaction per sample for data storage at public blockchains. As we pointed out in our previous publications [10,33], it is better to first aggregate sensing data and only place the aggregated data on the blockchains for immutability protection.

Even with aggregation, the data for each sample could be 500 bytes or larger. Considering that Ethereum charges 20,000 gas for each 32-byte of data, this means that the minimum cost for each data entry of 500 bytes would be 312,500 gas. Assuming that the gas price is 14.3 gwei (1 gwei = 10^{-9} ETH) and the ETH price is USD 1640, this transaction would cost at least USD 7.33. While this amount does not appear to be large, the total transaction fees would add up in the long run. For example, if one transaction is issued every 10 minutes, 144 transactions will be issued every 24 h and the total cost in transaction fees would be over USD 1000 per day! Only large institutions could afford this amount of transaction fees.

To address this issue, we propose to store the full data entry in IPFS and record the IPFS `hash` returned by the IPFS in the smart contract. The IPFS `hash` is a 46-byte-long self-describing multihash that can be used to identity both the content of the data entry as well as the peer and key to access the entry in the IPFS network (https://richardschneider.github.io/net-ipfs-core/articles/multihash.html (accessed on 1 March 2023)). The mechanism to do so is illustrated in Figure 9. With this design, only 46 bytes are stored in the smart contract instead of 500 bytes. For each transaction, this would saves 272,500 gas, which is equivalent to USD 6.39, and the daily savings would be USD 920.26.

4.2. Secure Genome Analysis

In the context of secure genome analysis as defined in [36,37], each observation is fairly short, hence, it is acceptable to store all the observations directly in the smart contract. However, we show here that it is unnecessary to store extra keys with wildcards in one or more mappings to facilitate queries with one or more wildcards.

For secure genome analysis, the gene, variant, and drug are used together as the key to each observation. In our smart contract, the gene–variant–drug combination is encoded into "gene/variant/drug" as the tag to each observation. In this use case, the tag is unique to each observation. The data entry for each observation is encoded in the form "gene/variant/drug/outcome/relation/sideeffect." We use the symbol "/" as the separator

because the gene name may contain "-". A string encoded this way can be stringified easily in all modern programming languages.

For a query using the full gene–variant–drug combination, the corresponding observation can be retrieved directly via the READ(tag) function. For a query with one or more wildcards, all data entries are first retrieved using the READALL() function. Once the dataset is retrieved, further queries with wildcards will not induce any call to the smart contract. Each entry in the dataset is then transformed from one string into six strings (gene, variant, drug, outcome, relation, side effect). A query with one or more wildcards will entail a linear search of the dataset, which is identical to the solutions proposed in [36,37].

4.3. Security Analysis of the Proposed Datastore

We analyze the security of the proposed datastore with respect to the three primary security properties: confidentiality, integrity, and availability [43]. We also discuss the risk of privacy leakage. Confidentiality means that the data are not revealed to the general public, i.e., only those who are authorized may access the data. Integrity means that the data come from a trusted source, and are kept intact in their original form, i.e., the data are not falsified and not modified in any unauthorized way. Availability means that the data are accessible to authorized users whenever needed. Privacy is closely related to confidentiality and it typically emphasizes the protection of the identity of the owner of the data. Privacy leakage refers to the violation of both the confidentiality and the privacy of the data, i.e., unauthorized users (such as adversaries) may obtain sensitive data from a system and somehow could learn the identity of the data owners [44].

Confidentiality is protected via strong user authentication so that unauthorized users cannot access the data stored in a system. In the proposed system, the data stored in the smart contract are accessible only to the smart contract owner and those who have been granted a read, write, or admin privilege. Hence, the proposed system guarantees data confidentiality.

The protection of the data integrity in a system requires more complex mechanisms. First, a mechanism must exist to accept data from legitimate sources only. Second, once the data are stored in the system, an additional mechanism must be in place to prevent unauthorized modification of the data or the deletion of the data. In the proposed system, only the smart contract owner and the users that have been granted the write or admin privilege may insert data to the system. Assuming that the smart contract owner is trusted, then, the system ensures that all data come from legitimate sources. Since the blockchain guarantees data immutability, once data are inserted into the smart contract, they cannot be modified or deleted. Therefore, the proposed system ensures the integrity of the data.

Availability is protected by both system fault tolerance and proper authentication of users. Fault tolerance ensures that the data would exist for access at any time. Proper authentication would grant authorized users access to the data whenever requested. Since the proposed system depends on the blockchain and IPFS, both have built-in fault tolerance mechanisms with heavy degree of replication of the data to ensure that the data are always available. The user authentication mechanism as part of the proposed smart contract would ensure all authorized users have access to the data. Hence, the proposed system ensures availability of the data.

Next, we argue that the proposed system has low risk of privacy leakage. As the proposed system is built on top of Ethereum, each of the users is identified by an address. This design grants all users of the proposed system pseudo-anonymity. Furthermore, the user access control mechanism incorporated into the smart contract ensures that an unauthorized adversary does not have access to the data stored in the system. Hence, neither the identity (other than the address) of the user, nor the data are revealed to adversaries.

We note that the security of all blockchain systems relies on the proper protection of the private keys. Once the private key is stolen, the corresponding account can be compromised. On the other hand, if the private key is lost, the corresponding account

would not be able to generate new transactions. Should the latter scenario happen, the data placed in the secure database would no longer be accessible by this account, and this account would not be able to make contract write calls either (if it has been granted the write or admin privilege).

Finally, before we conclude the security analysis of the proposed datastore, it is necessary to mention the potential risk of IPFS. Although virtually all academic publications (such as [45,46]) have assumed that IPFS is secure, there are known vulnerabilities in IPFS (https://consensys.net/diligence/blog/2022/09/the-forgotten-ipfs-vulnerabilities/ (accessed on 1 March 2023)).

5. Experimental Results and Discussion

The primary objective of the experiments is to demonstrate that our smart contract has the advantage of incurring low gas consumption with sufficient expressibility to accommodate different application scenarios. In addition, we fully characterize the performance of the smart contract with and without IPFS integration. Unlike [36,37], we focus on using gas consumption as the metric for evaluating the efficiency of the smart contract design instead of runtime latency for write and read. Since, by design, public blockchain systems must target a sufficiently long block interval for the decentralized consensus to complete, the time it takes to write to blockchain inevitably depends on the target block interval.

The experiments were performed using an iMac-27 with core-i5 CPU and 64 GB RAM. The truffle suite (https://trufflesuite.com (accessed on 1 March 2023)) was used for development and testing. The truffle suite consists of the truffle tools (truffle compile, truffle migrate, and truffle console), and Ganache, which runs a local Ethereum node to facilitate the development, deployment, and testing of smart contracts.

In our implementation, we chose to use the Python language to be consistent with our previous work on securing sensing data processing and logging [13]. A Python script is developed to compile and deploy our Ethereum smart contract to a local Ganache node. The script also implements the logic to communicate with the local IPFS node, and to issue signed transactions to the Ganache node. The script imports a number of libraries to work with Ethereum smart contracts and with IPFS, including py-solc-x (https://pypi.org/project/py-solc-x/ (accessed on 1 March 2023)), web3.py (https://pypi.org/project/web3/ (accessed on 1 March 2023)), and py-ipfs-http-client (https://pypi.org/project/ipfshttpclient/ (accessed on 1 March 2023)).

5.1. Comparison with Competing Approaches

To compare with existing smart contract solutions for secure data storage as reported in [36,37], we intentionally removed user access control from our smart contract so that the comparison would be fair because the smart contracts introduced in [36,37] have no user access control. Removing user access control eliminates possible complexity introduced by user access control during the data storage and query operations.

In the comparison, we use four observations from the secure genome analysis competition dataset as the inputs for the tests. The input consists of six components: gene name, variant, drug name, outcome (improved, unchanged, or deteriorated), possible relation (yes or no), side effect (yes or no):

- case1: (“HLA-B”, “57”, “abacavir”, “improved”, yes, no);
- case2: (“CYP3A5”, “52”, “pegloticase”, “unchanged”, yes, yes);
- case3: “CYP3A5”, “53”, “pegloticase”, “unchanged”, yes, yes);
- case4: “CYP3A5”, “53”, “abacavir”, “unchanged”, yes, yes).

We compare our solution with the four smart contract solutions designed for secure gene analysis, in the order of the ranking from the first place winner to the honorable mention during the 2019 competition [36,37]: query index [36], index everything [36], full index [37], and dual-scenario indexing [36] for the four cases. The results are summarized in Table 1. The mean gas consumption for different studies is illustrated in Figure 10.

Table 1. Comparison between our smart contract and competing solutions in terms of the gas consumed for the write operation.

Input	This Study	Query Index	Index Everything	Full Indexing	Dual Scenario Indexing
deploy	823,933	903,780	707,460	896,850	791,741
case1	160,706	345,226	79,394	238,526	339,863
case2	146,526	453,249	84,818	333,549	339,902
case3	146,490	401,949	50,618	238,646	339,902
case4	129,354	384,786	50,582	204,374	339,863
mean	145,769	383,478	66,353	253,774	339,883

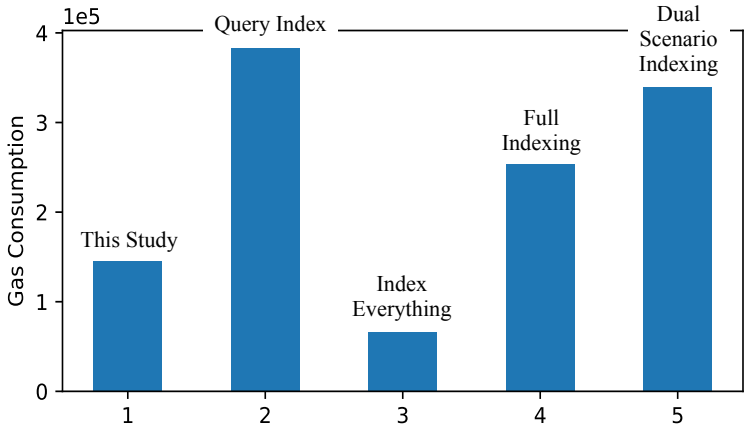

Figure 10. Comparison of gas consumption for different studies.

Prior to testing the gas consumption of the write operation, the smart contract for the respective solutions is first deployed. The deployment cost of the smart contract is shown in the first data row of Table 1. While the deployment cost is high in all solutions, they differ very little.

We note that the four cases are used as the input to the WRITE function of the smart contract in order (i.e., case1 is followed by case2, case3, and case4), and the order matters for the competing solutions as explained below. In our smart contract, the WRITE functions take two arguments—a tag and the data entry, both in the string format. For the four cases, we concatenate the first three components in the form of "c1/c2/c3" as the tag and concatenate all six components in the form of "c1/c2/c3/c4/c5/c6" as the data entry. As the first three parameters' (gene-variant-drug) combinations are all different in the four cases, the tags are all different, which means that in each case, the mapping would create a new entry and the gas consumption would be determined by the tag and data entry length.

It is not surprising that the gas consumption for the query index solution is much higher than ours (more than twice of ours). Although the query index solution also uses a single mapping and a single string array to store the data, multiple entries are created in the mapping in each of the cases. In our solution, only a single entry is created in the mapping. More specifically, in case1, the query index solution creates a total of eight entries in the mapping when storing the observation (one entry with a unique gene–variant–drug combination and seven entries with one or more wildcards). In case2, the query index creates a total of seven entries in the mapping because the case of "*/*/*" has already been created during case1 and this existing entry is updated. The higher gas consumption for case2 is due to the longer drug name and longer outcome strings. In case3, five entries are added into the mapping due to the variant change between case2 and case3 and three existing entries are updated. As a result, the gas consumption is about 50,000 gas less in case3 than in case2. In case4, only a single entry is created in the mapping and seven

existing entries are updated, which is why the gas consumption is the smallest among the four cases.

The gas consumption for the index everything solution is much smaller than that for our solution. This is because the index everything solution is customized for the dataset used in the secure gene analysis competition where the gene names, variants, and drug names are encoded as 8-bit integers, and the gene–variant–drug combination is encoded as 24-bit integers. Five mappings are used to map each gene-variant-drug to the number of presence of each outcome/relation/side effect (using a 32-bit integer). As such, the mappings take little space. In all cases, the key of the mapping takes one 32-byte word, and the value of the mapping takes another 32-byte word. We note that while this solution works well for the secure gene analysis competition, it is not a good fit for general purpose data storage.

The gas consumption for the full indexing solution is between 50% to 100% higher than that for our solution. This is due to the use of three mappings compared to a single mapping in ours. In case1, one new entry is created for each of the three mappings. In case2, one new entry is also created for each of the three mappings. The reason why the gas consumption for case2 is higher is because the gene name and the drug name are longer than those in case1. In case3, a new entry is created in the variant mapping and one entry from each of the gene and drug mappings is updated, and the same is true for case4. Updating an existing entry in a mapping costs less than creating a new entry.

The gas consumption for the dual-scenario indexing is consistently high in all four cases due to the use of a large structure to store the observation (i.e., `GeneDrugRelation`). This solution uses two mappings. One mapping (`GeneDrugRelationKeyMapping`) is created and populated prior to any query can be accepted. The `GeneDrugRelationKeyMapping` maps all possible gene–variant–drug combinations with one or more wildcards to the list of full gene-variant-drug (without any wildcard) combinations. The other mapping (called `geneData`) is used to map full gene-variant-drug to the `GeneDrugRelation` record. The WRITE function only updates the `geneData`. Hence, the reported gas consumption in Table 1 does not include the cost for populating the `GeneDrugRelationKeyMapping`. Due to the way the data are stored (i.e., using a struct instead of a string), this solution is not extensible for general purpose data storage.

5.2. Gas Consumption w.r.t Data Entry Size

It is well known that Ethereum uses 32-byte words to store data, and the byte32 type is the preferred type to minimize gas consumption. However, it is much more convenient to use string to store data. In this section, we report the gas consumption versus different-sized data entries encoded as strings. During the experiments, we vary the data entry size from 1 to 640 bytes with a 1-byte increment in a loop to call the WRITE function defined in the smart contract. We experimented with four scenarios: (1) no user access control and the same tag for all data entries; (2) no user access control and each data entry is associated with a unique tag; (3) with user access control and the same tag for all data entries; (4) with user access control and each data entry is associated with a unique tag. In Scenarios (1) and (3), the string literal "tag1" is used for all data entries. In Scenarios (2) and (4), the tag is constructed by concatenating "tag" and the loop index, which changes from 0 up to 639. The experimental results are shown in Figure 11.

As can be seen in Figure 11a, the gas consumption increases linearly with the size of the data entries for Scenario (1). The results for Scenarios (2), (3), and (4) look very similar in this scale. To show the subtle differences between the four scenarios and also show the details on how the gas consumption increases with respect to the size of the data entries, we include the charts for a very limited range of the data entry sizes between 1 to 69 bytes in Figure 11a,b. As can been seen, the very first call to the WRITE function incurs higher gas than subsequent calls. The first call incurs 115,682 gas for Scenarios (1) and (2) and 118,287 for Scenarios (3) and (4). The next call incurs 84,294 for Scenario (1), 101,394 for Scenario (2), 86,899 for Scenario (3), and 103,999 for Scenario (4).

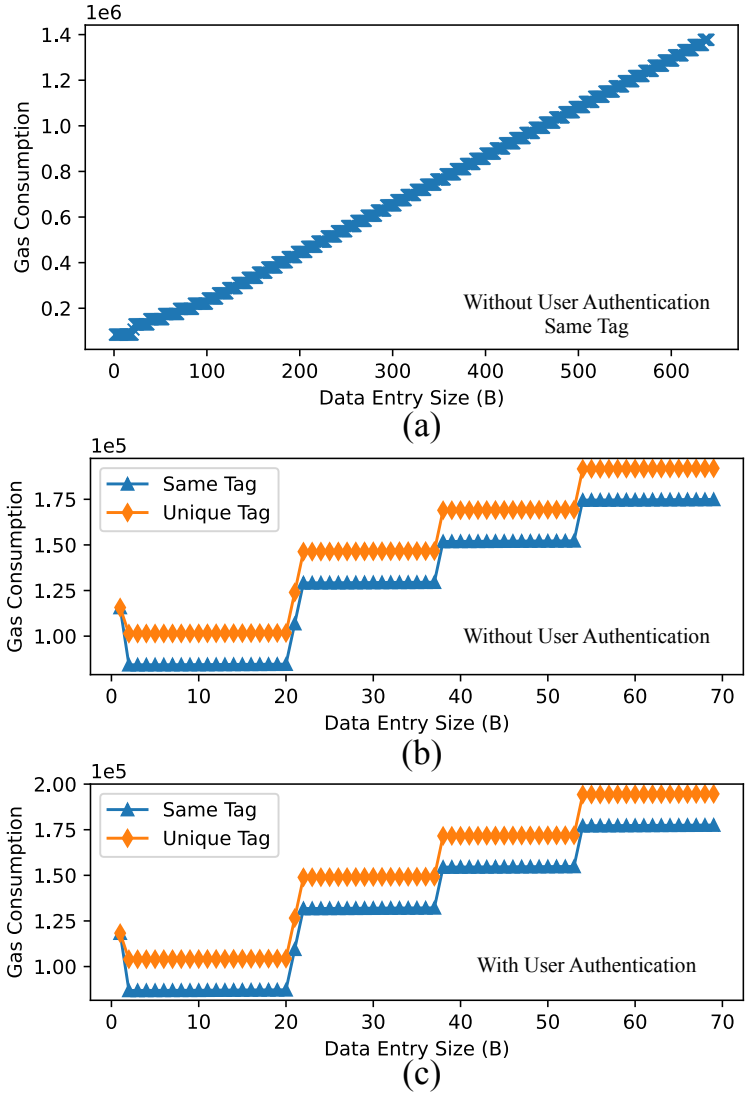

Figure 11. Gas consumption with respect to the size of the data entries (**a**–**c**).

It is unclear why in all scenarios the gas consumption has a jump from size 20 to size 21 (with 22,227 gas) and immediately has another jump from size 21 to size 22 (with 22,328 gas). After this initial period, there is a jump in gas consumption (23,328 gas) for every 16 bytes in the data entry size. A further examination shows that the gas consumption increases 24 gas for each additional byte in the size of the data entry. In Ethereum smart contract, string is encoded using the utf-8 variable-length character encoding standard where each string character is encoded using one to four bytes. Based on our observation, it looks like each string character is encoded using two bytes, which is why there is a jump in gas consumption for every 16 string characters.

The gas consumption for Scenario (2) is 17,100 higher than that for Scenario (1) for the same data entry size. The same is the case for Scenario (4) versus Scenario (3). This shows that the cost of creating a new entry in the tag mapping is 17,100 gas. The gas consumption for Scenario (3) is 2605 higher than that for Scenario (1) for the same data entry size. The same is the true for Scenario (4) compared with Scenario (2). This means that the cost of user authentication is only 2605 gas.

5.3. Gas Consumption for User Management

In this section, we report the gas consumption for user management operations. With user management and user access control, the deployment cost for our smart contract is 2,742,751 gas, which is significantly higher than without. Table 2 shows the gas consumption for grant roles (GRANTADMIN, GRANTWRITER, GRANTREADER) and delete roles (DELETEADMIN, DELETEWRITER, DELETEREADER) operations. As can be seen, the most expensive operation is to grant a new user the `Admin` role, which costs 95,198 gas because three entries in the `roles` mapping are updated. It costs a little less to grant a new user with the `Write` role, at 72,867 gas, where two entries in the `roles` mapping are updated. It costs the least to grant a new user with the `Read` role, at 49,977 gas, where only a single entry in the `roles` mapping is updated.

Table 2. Gas consumption for user management operations.

Role	GRANTROLE	DELETEROLE
Admin	95,198	37,262
Write	72,867	31,347
Read	49,977	28,613

An account may be stripped off the Admin, Writer, or Reader role via the DELETEADMIN, DELETEWRITER, DELETEREADER functions. Only the account that created the smart contract (i.e., the Owner) may remove users with the Admin role. Accounts with the write or read roles may be deleted by any account with the Admin role. The DELETEADMIN incurs 37,262 gas. The DELETEWRITER incurs 31,347 gas. The DELETEREADER incurs 28,613 gas. The gas consumption for user management is summarized in Table 2 and illustrated in Figure 12. As expected, the gas consumption for addition and deletion of admin accounts is higher than those of the writer accounts, which in turn is higher than those of the reader accounts.

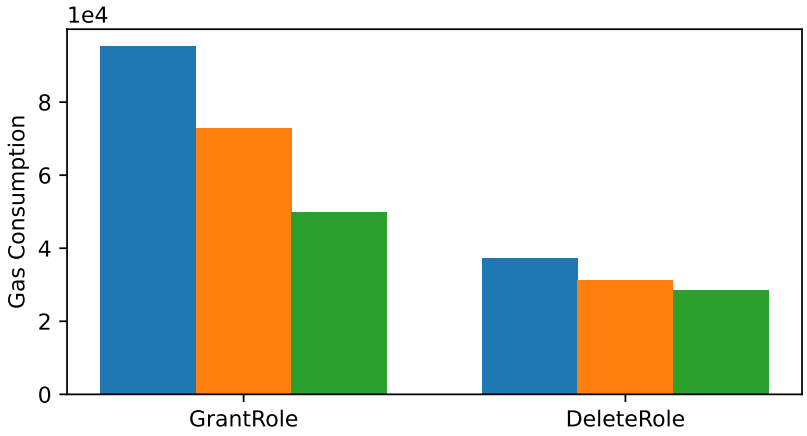

Figure 12. Gas consumption for granting roles and deleting roles.

5.4. Data Storage and Retrieval with IPFS and Smart Contract

In this section, we report the performance of the wrapper functions and demonstrate the advantage of integrating IPFS and smart contract as described in Section 4.1. We measure the latency for uploading and downloading data to/from IPFS and the latency in issuing write/read transactions to the smart contract (with user access control). Additionally, we also measure the gas consumption for the write and read transactions to our smart contract. We experimented with two scenarios: (1) storing only the IPFS `hash` with a tag to the smart contract (using the wrapper functions illustrated in Figure 9); and (2) storing the entire data with a tag to the smart contract (without using the wrapper

functions). During the experiments, we vary the data size from 100 to 2000 bytes with a step of 100 bytes. The transactions use distinct tags for subsequent transactions.

The latency results for the write and read transactions on the smart contract are illustrated in Figure 13. The curves labeled with "Contract Write" and "Contract Read" are the results for Scenario (1) where only the IPFS `hash` is stored at the smart contract. The curves labeled with "Contract Write Full" and "Contract Read Full" are the results for Scenario (2) where the entire data entry is stored at the smart contract without the integration with IPFS. As can be seen, the latency for the write/read transactions generally increases with the data entry size in Scenario (2). The latency for the write/read transactions in Scenario (1) remains constant and lower than that for Scenario (2). These results are as expected because in Scenario (1) the transaction size is the same for all original data entry sizes, and in Scenario (2) the write transaction size and the transaction receipt for the read transaction increase with the data entry sizes.

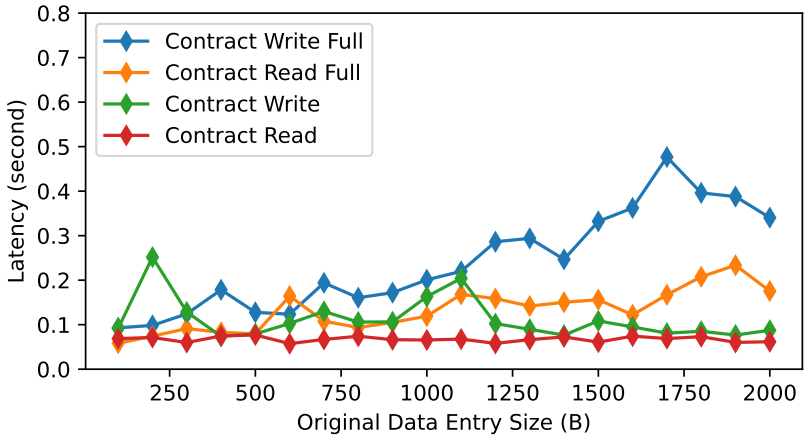

Figure 13. Latency results in interaction with the smart contract.

As can be seen in Figure 13, the latency is below 0.5 s for all scenarios and for all data entry sizes. We regard this latency satisfactory because the transaction rate must be kept relatively low by considering several factors: (1) there is a very limited system throughput capability due to the need for reaching decentralized consensus in blockchain (in Ethereum, the target block interval is 12 s and there are usually less than 500 transactions per block); (2) transactions that mutate the blockchain state (i.e., contract write) would incur significant financial cost and even read-only transaction submitted to the blockchain (i.e., not served locally) would still incur the base transaction gas fee. As such, users should only store the most critical data in the smart contract and the calls on the smart contract should be made infrequent, for example, at most a few times per hour. The observed latency results in our experiments mean that our system can sustain a 1Hz transaction generation rate (i.e., one transaction per second), which is much higher than that recommended for practical uses.

The latency results for IPFS uploading (i.e., `add_json()`) and downloading (i.e., `get_jsoon()`) are shown in Figure 14. Downloading data from IPFS incurs the least latency (0.002–0.003 s), and the latency remains small for all data sizes. However, the latency for uploading data to IPFS is much larger (around 0.1 s) and has huge jitters for the first few times. Again, this is expected because uploading would need to push the data first to the local IPFS node, and then to a few nearby IPFS nodes, which takes time, and downloading can be performed with the local IPFS node.

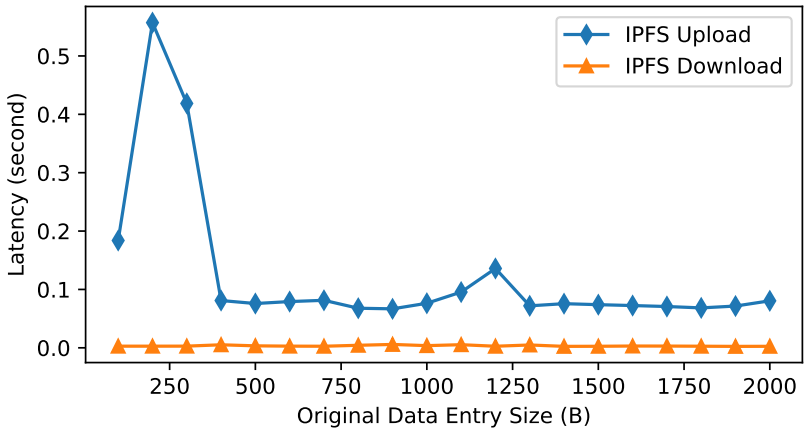

Figure 14. Latency in interaction with IPFS.

The gas consumption results for the two scenarios are shown in Figure 15. As can be seen, for Scenario (2), the gas consumption increases linearly with the data entry sizes for both the write and read transactions. The rate of increase is much larger for the write transactions than that for the read transactions (i.e., the two curves labeled "Contract Write/Read Full"). In contrast, the gas consumption for Scenario (1) for both the write and read transactions remains constant (except the very first transaction during the run), where the gas consumption for the write transactions is much lower than that for Scenario (2) as the data entry sizes increase, and the gas consumption for the read transactions is also lower than that for Scenario (2) because the returned data are shorter. These results demonstrate that it is advantageous to use the proposed approach (in terms of financial cost savings) that integrates with IPFS and stores only the IPFS `hash` of the data stored in IPFS in the smart contract.

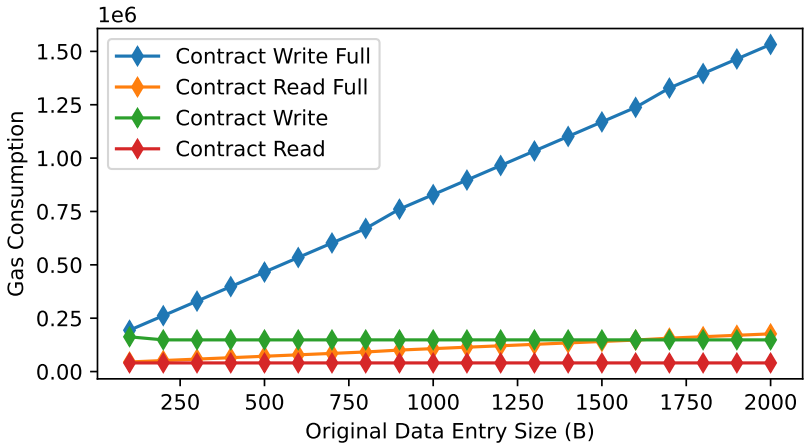

Figure 15. Gas consumption with and without IPFS integration.

It is also informative to compare the total latency for read and write of data in Scenario (1) and Scenario (2). The ratio of the total latency between Scenario (1) and Scenario (2) is shown in Figure 16. If the ratio is below 1, it means that the total latency for IPFS read/write and contract read/write for the integrated approach (i.e., Scenario (1) is lower than the corresponding latency for Scenario (2). As can be seen, the ratio is below 1 consistently if the original data entry size is 1250 bytes or higher.

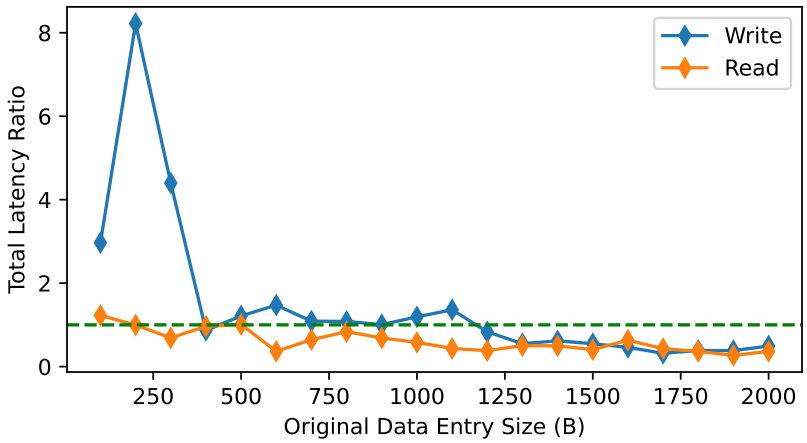

Figure 16. Total latency ratio for the read and write operations between Scenario (1) and Scenario (2).

5.5. Discussion

We emphasize here that blockchain-based secure datastore should not be regarded as a secure version of traditional database system. Unlike traditional database systems, which could offer a transaction rate higher than 1000 per second, blockchains are designed to have limited throughput due to the need to allocate sufficient time interval for reaching decentralized consensus. There are two conflicting factors: (1) target block interval, and (2) upper limit on the block size. A smaller block interval and a larger block size would be conducive to achieving higher system throughput. Unfortunately, the use of a smaller block interval would inevitably necessitate the use of a smaller block size for system stability [47].

For example, Ethereum currently has a target block gas limit of 15 million gas with maximum limit of 30 million gas. The minimum gas consumption for a transaction is 21,000 gas. This means that the target number of transaction per block is 714 (maximum number of transactions is 1428). The current target block interval is 12 s. This means that the target system throughput would be 59 transactions per second (maximum 118 transactions per second). Considering that Ethereum has thousands of users, we recommend that users of our proposed datastore generate less than one transaction per minute for contract write. If users must handle data generated with higher frequency, we recommend aggregating the raw data first, as we have proposed in [13]. If aggregating data is not possible, then our proposed secure datastore is not suitable for the application because making too-frequent contract write calls would create a backlog and severely increase the latency to an unacceptable level.

Another limiting factor for our secure datastore is the financial cost. Even with infrequent contract write and using the smart contract to record IPFS `hash` only, the system would still incur significant financial cost in the long run. Using the gas cost reported in Figure 15, a contract write call costs about $0.2 \times 10^6 = 200,000$ gas. If one contract write call is made per hour, the total daily gas consumption would be $200,000 \times 24 = 4,800,000$. Again, using the gas price of 14.3 gwei and the ETH price of USD 1640, the daily cost would be USD 2.57 and the annual cost would be USD 41,087.9. Not all organizations can sustain such high financial cost.

6. Conclusions

This article presented the details of the design and implementation of a secure datastore based on Ethereum smart contract. The datastore has built-in user access control. While the smart contract itself is publicly visible, only authorized users may write data entry to the datastore and only authorized users may read from the database. Furthermore, the datastore allows tag-based query of the data stored in the datastore.

To facilitate user access control, three different roles are defined in the smart contract. The account that created the smart contract (i.e., the owner of the smart contract) has the

highest privilege and it may grant an account the administrator, writer, or reader role. The account that has administrative privilege may grant other accounts the writer or reader roles. The account that has the write or administrative privilege may write data entries with a tag to the datastore. The account that has the read, write, or administrative privilege may retrieve data entries in the datastore. We argue that the proposed tag-based query is simple, yet flexible enough to facilitate complex queries.

Other than these two features, the primary consideration of the datastore design is to minimize gas consumption of the smart contract. To demonstrate the efficiency in gas consumption of our smart contract design, we compared it with four competing solutions. We showed that our solution incurs the second-smallest gas consumption. The solution that incurred lower gas consumption than ours was custom-made for the secure gene analysis competition and it is not usable for general-purpose data storage and retrieval.

Furthermore, we recommend to store the original data entry in IPFS and only store the IPFS `hash` of the data entry in the smart contract. We demonstrated that the proposed integration with IPFS can significant reduce the gas consumption compared with storing all the data entry in the smart contract. Furthermore, we showed experimentally that if the data entry size is 1250 bytes or larger, the IPFS-smart contract integrated solution would incur lower latency for writing to and reading from our datastore compared with storing the entire data entry in the smart contract.

Limitations of the Approach

Pros. The proposed tag-based datastore aimed to provide the right tradeoff for query flexibility, data confidentiality, smart contract security, and minimizing the financial cost of the datastore. The advantages of the proposed datastore includes its simplicity, flexibility, user access control, and user management. The simplicity of the smart contract design minimizes the likelihood of introducing security vulnerabilities and reduces the gas consumption of its operations. The flexibility of the design supports various applications (such as sensing data logging and genome analysis) and use cases (such as storing plain or encrypted tag and data).

Cons. The obvious limitation of the proposed datastore is that it relies on Ethereum. Additionally, the emphasis on simplicity would inevitably force the users to perform additional operations, such as encryption and extraction of the needed answer from the returned result of a datastore read. The current system does not yet offer a graphical user interface. Furthermore, the number of applications and use cases is relatively limited.

In the future, we plan to improve the current design and system implementation by addressing these limitations. First, we will compile more applications of the proposed secure datastore beyond sensing data logging and genome analysis. Second, we will enhance the datastore frontend with more wrapper functions to reduce the burden on the users (i.e., data producers and data consumers). For each type of application, some wrapper functions could be developed such that the datastore resembles more a traditional database system. For example, the data insertion and query functions similar to what has been used in the genome analysis could be offered by the database frontend. Internally, the calls would be transformed so that our tag-based smart contract can be used. Third, we will port the tag-based smart contract from Ethereum to other blockchain platforms (such as Hyperledger) that support Turing-Complete smart contracts. Finally, we will enhance the research prototype with a graphic user interface and a suite of examples in Python and JavaScript. We plan to share our implementation source code as a public GitHub project.

Author Contributions: Conceptualization, W.Z., H.U. and L.L.; methodology, W.Z.; software, I.M.A. and W.Z.; validation, I.M.A., W.Z., H.U. and L.L.; formal analysis, W.Z.; investigation, I.M.A. and W.Z.; resources, W.Z., H.U. and L.L.; data curation, I.M.A. and W.Z.; writing—original draft preparation, I.M.A. and W.Z.; writing—review and editing, W.Z.; visualization, W.Z.; supervision, L.L.; project administration, H.U. and L.L.; funding acquisition, H.U. and L.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by US Department of Energy grant number DE-FE0031745.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: We would like to thank the anonymous reviewers for their highly constructive criticisms and invaluable suggestions. This work was supported by the United States Department of Energy Award DE-FE0031745.

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

Appendix A

The Solidity source code for the proposed smart contract is provided in the appendix.

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.6.0 <0.9.0;
pragma experimental ABIEncoderV2;

contract datastore {
  address public  owner;

  event GrantRole(bytes32 indexed role, address indexed account);
  event RemoveRole(bytes32 indexed role, address indexed account);

  constructor(){
    owner=msg.sender;
    _grantRole(Admin, msg.sender);
  }

  mapping(string => uint []) public tagindex;
  string [] public database;
  uint counter=0;
  mapping(bytes32 =>mapping(address=>bool)) public roles;

  bytes32 private constant Admin = keccak256(abi.encodePacked("Admin"));
  bytes32 public constant Read = keccak256(abi.encodePacked("Read"));
  bytes32 public constant Write = keccak256(abi.encodePacked("Write"));

  modifier onlyRole (bytes32 _role) {
    require(roles[_role][msg.sender], "not authorized");
    _;
  }

  function _grantRole(bytes32 _role, address _account) internal {
    if (_role==Admin) {
      roles[Admin][_account]=true;
      roles[Write][_account]=true;
      roles[Read][_account]=true;
    } else if(_role==Write) {
      roles[Write][_account]=true;
      roles[Read][_account]=true;
    } else{
      roles[_role][_account]=true;
    }

    emit GrantRole(_role, _account);
  }

  function grantAdminRole(address _account) external onlyRole(Admin){
    require(_account != address(0), "Roles: account is the zero address");
    require(!roles[Admin][_account], "the Role with address is  already
        available");
    if (msg.sender == owner) {
      _grantRole(Admin, _account);
    } else {
     revert (" admin not authorized to grant Admin Role");
```

```
52      }
53    }
54
55    function grantWriteRole(address _account) external onlyRole(Admin){
56      require(_account != address(0), "Roles: account is the zero address");
57      require(!roles[Write][_account], "the Role with address is  already
            available"); //
58      _grantRole(Write, _account);
59    }
60
61    function grantReadRole(address _account) external onlyRole(Admin){
62      require(_account != address(0), "Roles: account is the zero address");
63      require(!roles[Read][_account], "the Role with address is  already
            available"); //
64      _grantRole(Read, _account);
65    }
66
67    function deleteAdminRole(address _account) external onlyRole(Admin) {
68      require(roles[Admin][_account], "the Role with address is not
            available to be deleted"); // check if the address available
69      if (msg.sender == owner) {
70        deleteRole(Admin,_account);
71      } else {
72          revert (" admin not authorized to delete owner");
73      }
74    }
75
76    function deleteWriteRole(address _account) external onlyRole(Admin) {
77      require(roles[Write][_account], "the Role with address is not
            available to be deleted"); // check if the address available
78      deleteRole(Write,_account);
79    }
80
81    function deleteReadRole(address _account) external onlyRole(Admin) {
82      require(roles[Read][_account], "the Role with address is not available
             to be deleted"); // check if the address available
83      deleteRole(Read,_account);
84    }
85
86    function deleteRole(bytes32 _role, address _account) internal onlyRole(
        Admin) {
87      if(_role==Admin) {
88        roles[Admin][_account]=false;// false means deactive the content
              means delete
89        roles[Write][_account]=false;
90        roles[Read][_account]=false;
91      } else if(_role==Write) {
92        roles[Write][_account]=false;
93        roles[Read][_account]=false;
94      } else {
95        roles[_role][_account]=false;
96      }
97      emit RemoveRole(_role, _account);
98    }
99
100   function write(string memory tag, string memory data) public onlyRole(
        Write) {
101     database.push(data);
102     tagindex[tag].push(counter);
103     counter++;
104   }
105
106   function read(string memory tag) public onlyRole(Read) view returns (
        string [] memory) {
107     uint [] memory indices = tagindex[tag];
108     uint length = indices.length;
109     string [] memory result = new string[](length);
110     for(uint i=0;i<length;i++){
111       string memory entry = database[indices[i]];
```

```
112         result[i] = entry;
113       }
114       return result;
115     }
116
117     function readall() public onlyRole(Read) view returns (string [] memory)
            {
118       return database;
119     }
120
121     function readrecent(uint n) public onlyRole(Read) view returns (string
            [] memory) {
122       if(n >= counter) {
123         return database;
124       }
125       string [] memory result = new string[](n);
126       for(uint i=0;i<n;i++){
127         string memory entry = database[counter-i-1];
128         result[i] = entry;
129       }
130       return result;
131     }
132
133     function getDatabaseSize() public onlyRole(Read) view returns (uint) {
134       return counter;
135     }
136 }
```

References

1. Denning, D.E.R. *Cryptography and Data Security*; Addison-Wesley Reading: Boston, MA, USA, 1982; Volume 112.
2. Yang, P.; Xiong, N.; Ren, J. Data security and privacy protection for cloud storage: A survey. *IEEE Access* **2020**, *8*, 131723–131740. [CrossRef]
3. Jatana, N.; Puri, S.; Ahuja, M.; Kathuria, I.; Gosain, D. A survey and comparison of relational and non-relational database. *Int. J. Eng. Res. Technol.* **2012**, *1*, 1–5.
4. Imran, S.; Hyder, I. Security issues in databases. In Proceedings of the 2009 Second International Conference on Future Information Technology and Management Engineering, Sanya, China, 13–14 December 2009 ; IEEE: Piscataway, NJ, USA , 2009; pp. 541–545.
5. Okman, L.; Gal-Oz, N.; Gonen, Y.; Gudes, E.; Abramov, J. Security issues in nosql databases. In Proceedings of the 2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications, Changsha, China, 16–18 November 2011; IEEE: Piscataway, NJ, USA, 2011; pp. 541–547.
6. Bertino, E.; Sandhu, R. Database security-concepts, approaches, and challenges. *IEEE Trans. Dependable Secur. Comput.* **2005**, *2*, 2–19. [CrossRef]
7. Zhao, W. *From Traditional Fault Tolerance to Blockchain*; John Wiley & Sons: Hoboken, NJ, USA, 2021.
8. Zhao, W. On Blockchain: Design Principle, Building Blocks, Core Innovations, and Misconceptions. *IEEE Syst. Man Cybern. Mag.* **2022**, *8*, 6–14. [CrossRef]
9. Wood, G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Proj. Yellow Pap.* **2014**, *151*, 1–32.
10. Zhao, W.; Upadhyay, H.; Lagos, L. Design and Implementation of a Blockchain-Enabled Secure Sensing Data Processing and Logging System. In Proceedings of the 2021 IEEE International Conference on Systems, Man, and Cybernetics (SMC), Melbourne, Australia, 17–20 October 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 386–391.
11. Zheng, G.; Gao, L.; Huang, L.; Guan, J. *Ethereum Smart Contract Development in Solidity*; Springer: Berlin/Heidelberg, Germany, 2021.
12. Kushwaha, S.S.; Joshi, S.; Singh, D.; Kaur, M.; Lee, H.N. Systematic review of security vulnerabilities in ethereum blockchain smart contract. *IEEE Access* **2022**, *10*, 6605–6621. [CrossRef]
13. Zhao, W.; Aldyaflah, I.M.; Gangwani, P.; Joshi, S.; Upadhyay, H.; Lagos, L. A Blockchain-Facilitated Secure Sensing Data Processing and Logging System. *IEEE Access* **2023**, *11*, 21712–21728. [CrossRef]
14. Li, R.; Song, T.; Mei, B.; Li, H.; Cheng, X.; Sun, L. Blockchain for large-scale internet of things data storage and protection. *IEEE Trans. Serv. Comput.* **2018**, *12*, 762–771. [CrossRef]
15. Liang, W.; Fan, Y.; Li, K.C.; Zhang, D.; Gaudiot, J.L. Secure data storage and recovery in industrial blockchain network environments. *IEEE Trans. Ind. Inform.* **2020**, *16*, 6543–6552. [CrossRef]
16. Popov, S.; Lu, Q. IOTA: Feeless and free. *IEEE Blockchain Tech. Briefs* **2019** .
17. Silvano, W.F.; Marcelino, R. Iota Tangle: A cryptocurrency to communicate Internet-of-Things data. *Future Gener. Comput. Syst.* **2020**, *112*, 307–319. [CrossRef]

18. Pinjala, S.K.; Sivalingam, K.M. DCACI: A decentralized lightweight capability based access control framework using IOTA for Internet of Things. In Proceedings of the 2019 IEEE 5th World Forum on Internet of Things (WF-IoT), Limerick, Ireland, 15–18 April 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 13–18.
19. Lamtzidis, O.; Gialelis, J. An IOTA based distributed sensor node system. In Proceedings of the 2018 IEEE Globecom Workshops (GC Wkshps), Limerick, Ireland, 15–18 April 2019; IEEE: Piscataway, NJ, USA, 2018; pp. 1–6.
20. Gangwani, P.; Perez-Pons, A.; Bhardwaj, T.; Upadhyay, H.; Joshi, S.; Lagos, L. Securing environmental IoT data using masked authentication messaging protocol in a DAG-based blockchain: IOTA tangle. *Future Int.* **2021**, *13*, 312. [CrossRef]
21. Suhail, S.; Hussain, R.; Khan, A.; Hong, C.S. Orchestrating product provenance story: When IOTA ecosystem meets electronics supply chain space. *Comput. Ind.* **2020**, *123*, 103334. [CrossRef]
22. Zheng, X.; Sun, S.; Mukkamala, R.R.; Vatrapu, R.; Ordieres-Meré, J. Accelerating health data sharing: A solution based on the internet of things and distributed ledger technologies. *J. Med. Inter. Res.* **2019**, *21*, e13583. [CrossRef] [PubMed]
23. Abdullah, S.; Arshad, J.; Khan, M.M.; Alazab, M.; Salah, K. PRISED tangle: A privacy-aware framework for smart healthcare data sharing using IOTA tangle. *Complex Intell. Syst.* **2022**, 1–19. [CrossRef]
24. Rydningen, E.S.; Åsberg, E.; Jaccheri, L.; Li, J. Advantages and opportunities of the IOTA Tangle for Health Data Management: A Systematic Mapping Study. In Proceedings of the 2022 IEEE/ACM 5th International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), Pittsburgh, PA, USA, 21–29 May 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 9–16.
25. Lücking, M.; Manke, R.; Schinle, M.; Kohout, L.; Nickel, S.; Stork, W. Decentralized patient-centric data management for sharing IoT data streams. In Proceedings of the 2020 International Conference on Omni-layer Intelligent Systems (COINS), Barcelona, Spain, 31 August–2 September 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 1–6.
26. Ordieres-Meré, J.; Villalba-Díez, J.; Zheng, X. Challenges and opportunities for publishing IIoT data in manufacturing as a service business. *Procedia Manuf.* **2019**, *39*, 185–193. [CrossRef]
27. Shih, C.S.; Yang, K.W. Design and implementation of distributed traceability system for smart factories based on blockchain technology. In Proceedings of the Conference on Research in Adaptive and Convergent Systems, Chongqing, China, 24–27 September 2019; pp. 181–188.
28. Lin, I.C.; Chang, C.C.; Chang, Y.S. Data Security and Preservation Mechanisms for Industrial Control Network Using IOTA. *Symmetry* **2022**, *14*, 237. [CrossRef]
29. Bhandary, M.; Parmar, M.; Ambawade, D. Securing Logs of a System-An IoTA Tangle Use Case. In Proceedings of the 2020 International Conference on Electronics and Sustainable Communication Systems (ICESC), Coimbatore, India, 2–4 July 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 697–702.
30. Shih, C.S.; Hsieh, W.Y.; Kao, C.L. Traceability for Vehicular Network Real-Time Messaging Based on Blockchain Technology. *J. Wirel. Mob. Netw. Ubiquitous Comput. Dependable Appl.* **2019**, *10*, 1–21.
31. Tesei, A.; Di Mauro, L.; Falcitelli, M.; Noto, S.; Pagano, P. IOTA-VPKI: A DLT-based and resource efficient vehicular public key infrastructure. In Proceedings of the 2018 IEEE 88th Vehicular Technology Conference (VTC-Fall), Chicago, IL, USA, 27–30 August 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 1–6.
32. Hrga, A.; Capuder, T.; Žarko, I.P. Decentralized IoT Platform for Flexibility Service Providers in Power Systems. In Proceedings of the 2021 IEEE International Conference on Blockchain (Blockchain), Melbourne, Australia, 6–8 December 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 1–7.
33. Zhao, W.; Yang, S.; Luo, X. Secure hierarchical processing and logging of sensing data and IoT events with blockchain. In Proceedings of the 2020 The 2nd International Conference on Blockchain Technology, Hilo, HI, USA, 12–14 March 2020; pp. 52–56.
34. Fauziah, Z.; Latifah, H.; Omar, X.; Khoirunisa, A.; Millah, S. Application of blockchain technology in smart contracts: A systematic literature review. *Aptisi Trans. Technopreneurship (ATT)* **2020**, *2*, 160–166. [CrossRef]
35. Gupta, R.; Tanwar, S.; Al-Turjman, F.; Italiya, P.; Nauman, A.; Kim, S.W. Smart Contract Privacy Protection Using AI in Cyber-Physical Systems: Tools, Techniques and Challenges. *IEEE Access* **2020**, *8*, 24746–24772. .10.1109/ACCESS.2020. 2970576. [CrossRef]
36. Kuo, T.T.; Bath, T.; Ma, S.; Pattengale, N.; Yang, M.; Cao, Y.; Hudson, C.M.; Kim, J.; Post, K.; Xiong, L.; et al. Benchmarking blockchain-based gene-drug interaction data sharing methods: A case study from the iDASH 2019 secure genome analysis competition blockchain track. *Int. J. Med. Inform.* **2021**, *154*, 104559. [CrossRef] [PubMed]
37. Gürsoy, G.; Brannon, C.M.; Gerstein, M. Using Ethereum blockchain to store and query pharmacogenomics data via smart contracts. *BMC Med. Genom.* **2020**, *13*, 1–11. [CrossRef] [PubMed]
38. Batchu, S.; Patel, K.; Henry, O.S.; Mohamed, A.; Agarwal, A.A.; Hundal, H.; Joshi, A.; Thoota, S.; Patel, U.K. Using ethereum smart contracts to store and share COVID-19 patient data. *Cureus* **2022**, *14*, e21378. [CrossRef]
39. Kurt Peker, Y.; Rodriguez, X.; Ericsson, J.; Lee, S.J.; Perez, A.J. A cost analysis of internet of things sensor data storage on blockchain via smart contracts. *Electronics* **2020**, *9*, 244. [CrossRef]
40. Priyadarshini, R.; Alagirisamy, M.; Rajendran, N. Medchain for Securing Data in Decentralized Healthcare System Using Dynamic Smart Contracts. In Proceedings of the Third International Conference on Image Processing and Capsule Networks: ICIPCN 2022, Bangkok, Thailand, 20–21 May 2022; Springer: Berlin/Heidelberg, Germany, 2022; pp. 574–586.
41. Manzoor, A.; Liyanage, M.; Braeke, A.; Kanhere, S.S.; Ylianttila, M. Blockchain based proxy re-encryption scheme for secure IoT data sharing. In Proceedings of the 2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), Seoul, Republic of Korea, 14–17 May 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 99–103.

42. Chen, L.; Lee, W.K.; Chang, C.C.; Choo, K.K.R.; Zhang, N. Blockchain based searchable encryption for electronic health record sharing. *Future Gener. Comput. Syst.* **2019**, *95*, 420–429. [CrossRef]
43. Pfleeger, C.P. *Security in Computing*; Prentice-Hall, Inc.: Hoboken, NJ, USA, 1988.
44. Krishnamurthy, B.; Naryshkin, K.; Wills, C. Privacy leakage vs. protection measures: The growing disconnect. In Proceedings of the Web 2.0 Security and Privacy, Oakland, CA, USA, 26 May 2011; IEEE: Piscataway, NJ, USA, 2011, Volume 2; pp. 1–10.
45. Nizamuddin, N.; Salah, K.; Azad, M.A.; Arshad, J.; Rehman, M. Decentralized document version control using ethereum blockchain and IPFS. *Comput. Electr. Eng.* **2019**, *76*, 183–197. [CrossRef]
46. Ali, M.S.; Dolui, K.; Antonelli, F. IoT data privacy via blockchains and IPFS. In Proceedings of the Seventh International Conference on the Internet of Things, Linz, Austria, 22–25 October 2017; pp. 1–7.
47. Akbari, E.; Zhao, W.; Yang, S.; Luo, X. The impact of block parameters on the throughput and security of blockchains. In Proceedings of the 2020 The 2nd International Conference on Blockchain Technology, Hilo, HI, USA, 12–14 March 2020; pp. 13–18.