

## Article

# Blockchain and Interplanetary File System (IPFS)-Based Data Storage System for Vehicular Networks with Keyword Search Capability

N. Sangeeta and Seung Yeob Nam \* 

Department of Information and Communication Engineering, Yeungnam University,  
Gyeongsan 38541, Republic of Korea

\* Correspondence: synam@ynu.ac.kr

**Abstract:** Closed-circuit television (CCTV) cameras and black boxes are indispensable for road safety and accident management. Visible highway surveillance cameras can promote safe driving habits while discouraging moving violations. According to CCTV laws, footage captured by roadside cameras must be securely stored, and authorized persons can access it. Footages collected by CCTV and Blackbox are usually saved to the camera's microSD card, the cloud, or hard drives locally but there are concerns about security and data integrity. These issues may be addressed by blockchain technology. The cost of storing data on the blockchain, on the other hand, is prohibitively expensive. We can have decentralized and cost-effective storage with the interplanetary file system (IPFS) project. It is a file-sharing protocol that stores and distributes data in a distributed file system. We propose a decentralized IPFS and blockchain-based application for distributed file storage. It is possible to upload various types of files into our decentralized application (DApp), and hashes of the uploaded files are permanently saved on the Ethereum blockchain with the help of smart contracts. Because it cannot be removed, it is immutable. By clicking on the file description, we can also view the file. DApp also includes a keyword search feature to assist us in quickly locating sensitive information. We used Ethers.js' smart contract event listener and contract.queryFilter to filter and read data from the blockchain. The smart contract events are then written to a text file for our DApp's keyword search functionality. Our experiment demonstrates that our DApp is resilient to system failure while preserving the transparency and integrity of data due to the immutability of blockchain.

**Keywords:** blockchain; Ethereum blockchain; decentralized application (DApp); interplanetary file system (IPFS); smart contracts



**Citation:** N. Sangeeta; Nam, S.Y. Blockchain and Interplanetary File System (IPFS)-Based Data Storage System for Vehicular Networks with Keyword Search Capability.

*Electronics* **2023**, *12*, 1545. <https://doi.org/10.3390/electronics12071545>

Academic Editor: Hamed Taherdoost

Received: 17 February 2023

Revised: 22 March 2023

Accepted: 22 March 2023

Published: 24 March 2023



**Copyright:** © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction and Background

CCTV camera images are a valuable source of traffic surveillance that supplements other traffic control measures. CCTV is aimed at helping in the detection and prevention of criminal activity. It can be helpful in protecting the citizens in the community. It is placed in public areas to provide evidence to appropriate law enforcement agencies. CCTV cameras can be found on busy roads, atop traffic lights, and at highway intersections. Operators detect and monitor traffic incidents using images from CCTV cameras. It may be possible to predict the duration of a traffic incident based on prior experience and traffic modeling techniques. Cameras are used to observe and monitor traffic, as well as to record traffic pattern data. Moving violation tickets are even issued using cameras.

The vehicle's event data recorder is constantly recording information in a loop while we are driving, at least until a collision occurs. Black boxes save data collected at the time of impact, as well as 5 s before and after the event. The black boxes will record all human contact with the vehicle. The data collected helps us understand the reasons for collisions and prevent them from happening again.

CCTV footage is being used in crime investigations by police officers and insurance companies [1] all over the world. Recorded footage is typically used by investigators to locate or confirm the identity of a suspect. Real-time surveillance systems allow employees or law enforcement officials to detect and monitor any threat in real time. Then, there's the archival footage record, which can be reviewed later if a crime or other issue is discovered. In these cases, the recorded footage must be securely deposited and kept for future use, making video storage a critical component of any video camera security system.

The vast majority of information collected by surveillance cameras and dashboard cameras is securely kept on hard drives as well as memory cards. The amount of storage on the MicroSD card of our security camera, on the other hand, is determined by the amount of activity recorded by our camera. This type of storage necessitates a large amount of storage space and exposes our data to risk if the device's hard drive fails or is damaged. It is critical to securely store CCTV and black box footage in order for it to be available and unaltered at all times. In many cases, the introduction and popularity of IP camera cloud storage have reduced the importance of local storage to a secondary option.

Cloud systems are an extremely good tool that offers us many advantages and functionalities. Cloud storage systems, on the other hand, have flaws such as problems with data safety [2,3], centralized data storage, and the requirement for trusted third parties. Owners are reassured of the burden of maintaining local data storage, but they end up losing direct control over storage reliability and protection. Every year, large database hacks cost millions of dollars. Furthermore, because the data is kept on an external device, the owners have no power over it; if the service provider disconnects or limits access, they will not be able to access it.

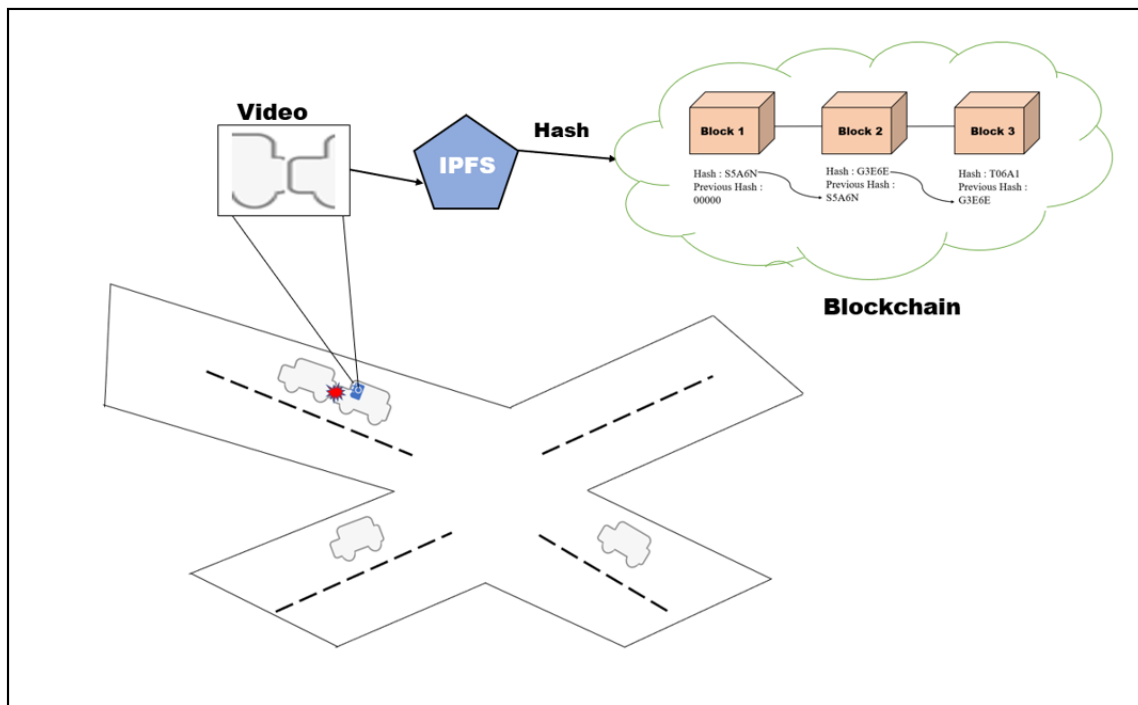
Due to the centralized nature of cloud storage data, an intruder to servers is able to view and alter it. Cloud data is untrustworthy and can be altered or removed at any time. As a result, making sure data security [4] and safeguarding users' privacy [5] are critical. Users are usually needed to cover the cost of any storage plan they select, even if they only use a portion of it.

Even the finest cloud service providers can face such a challenging problem while retaining strong maintenance standards. Centralized storage service providers occasionally fail to deliver the security service as agreed. For example, a hack on Dropbox [6] which is among the world's largest online storage companies, did result in the leak of 68 million usernames and passwords on the dark web. Well-known cloud services have started experiencing blackouts and security breaches. The mass email deletion event of Gmail [7], Amazon S3's latest shutdown [8], and the post-launch interruption of Apple MobileMe's [9] are other examples.

Blockchain technology may be able to address these issues. A blockchain is made up of a cryptographic algorithm, a timestamp, and transaction information that connects it to the preceding block. As a result, every block links to the next, forming a "chain" of blocks and producing safe and immutable records. In comparison, the blockchain is not designed for the purpose of file storage. The cost of keeping data on the blockchain is exorbitantly high. We can have decentralized as well as low-cost storage with the IPFS project [10]. Peer-to-peer networks provide greater security than centralized networks. As an outcome, they are ideal for protecting sensitive information from malicious actors.

We propose an IPFS-based distributed and decentralized storage application that offers more storage space compared to solely blockchain-based systems in this paper. Using distributed storage, information is kept on different nodes or servers on the Internet. To upload files, we use the Geth [11] software client to operate an Ethereum node and an IPFS Daemon server to operate our own IPFS node. Users will link to the DApp through the use of one's web browser as well as a blockchain wallet, Metamask [12], to connect to a blockchain in our proposed scheme. Since it is powered by Ethereum smart contracts, the decentralized application will interact with the blockchain, which will keep all the code of the application in addition to the data. The smart contracts keep track of all sources of information in IPFS files. A DApp can receive any kind of information. The hash value of the uploaded file is permanently saved on the Ethereum blockchain via smart contracts and

it cannot be changed or deleted. Whenever a file is uploaded, the DApp hears the event “File Upload” and updates the DApp’s user interface. We retrieve all of the smart contract events and reveal them on our DApp, which is called the “smart contract event log”. The smart contract event log contains data such as the file name, file summary (including the event and location of the file), file type, size of the file, time and date of upload, Ethereum account information of the user, and the hash value of the file once it has been uploaded to IPFS. Users can also view the file by clicking on its description. The user does not need to remember and save the hash value independently, which could be dangerous if another individual has access to it. Our DApp also includes a keyword search feature to assist you in quickly locating sensitive information. Figure 1 shows an example scenario where our proposed system can be applied. When an accident occurs, our proposed system might be used to save the video taken by the dashboard camera on IPFS and the hash value of the video on the blockchain to prevent the manipulation of the video using the immutability property of blockchain.



**Figure 1.** Example scenario to illustrate the application of the proposed system in the presence of accidents on the road.

The key contributions of our paper can be summarized as follows:

- Our proposed distributed storage application supports the storage of various file types since uploaded files are stored on IPFS and their hash values are stored in smart contracts on the Ethereum blockchain. Users need not remember the hash values since they can be retrieved from the blockchain later.
- DApp provides a keyword search feature to help users quickly find the necessary files based on Ethers.js’s smart contract event listener and contract.queryFilter.
- Our experiment shows that our DApp is resilient to system failure, and our system provides better transparency than is possible with centrally managed applications.

The rest of our paper is structured as follows: Section 2 contains related work. Section 3 contains preliminary information. The proposed scheme is described in Section 4. Section 5 goes over implementation. The performance evaluation results are described in Section 6. Finally, Section 7 brings the paper to a close.

## 2. Related Work

Hao, J. et al. [13] studied a blockchain and IPFS-based storage scheme for agricultural product tracking. During the manufacturing, processing, and logistics processes, sensors collect real-time data on product quality as well as video and picture data, according to this study. The server parses and encapsulates the data before writing it to IPFS, and the hash address is then stored in the blockchain to complete the data storage. The collected data is not directly written to an IPFS. The authors employ a private data server, and data collected by sensors is first stored on the private data server before being directly stored on the IPFS. If the server experiences problems, such as server failure, the collected data is lost, and the server is unable to write data to IPFS. There is no keyword search function for quickly finding agricultural product information.

Rajalakshmi et al. [14] proposed a framework for access control methods in research records that manages to combine blockchain, IPFS, as well as other traditional encryption methods. The system stores the verification metadata information acquired from the IPFS on the blockchain network using Ethereum smart contracts, resulting in tamper-proof record-keeping for further auditing. There is no keyword search functionality for searching information related to research records in this proposed scheme, which only stores PDF files.

Vimal, S. et al. [15] proposed a method to improve the efficiency of the P2P file-sharing system by incorporating trustworthiness and proximity awareness during file transfer using IPFS and blockchain. Any of these hashed files can be retrieved by simply calling the hash of the file. Miners who collaborate to ensure the successful transfer of resources are compensated. This study discusses the file transfer service, as well as the security strength and some of the IPFS-based incentives.

This system is built around IPFS and Blockchain. Yongle Chen et al. [16] proposed a more efficient P2P file system scheme. The authors pointed out the high-throughput problem for individual IPFS users by incorporating the responsibility of content service providers. A novel zigzag-based storage model is utilized to improve the IPFS block storage model by taking data reliability and availability, storage overhead, and other issues for service providers into account.

Rong Wang et al. proposed a video surveillance system relying on permissioned blockchains (BCs) and edge computing in their paper [17]. Convolutional neural networks (CNN), edge computing, and permissioned BCs, as well as IPFS technology, were used in this system. Edge computing was utilized to collect and process large amounts of wireless sensor data, while the IPFS storage solution was utilized to enable huge video data storage. CNN technology was applied to real-time monitoring, and Edge computing was utilized to gather and analyze large amounts of wireless sensor data.

Sun, J. et al. [18] proposed a blockchain-based secure storage and access scheme for electronic medical records in IPFS, which ensures necessary access to electronic medical data while preserving retrieval efficiency. IPFS is a file system used in order to store encrypted electronic medical data. After receiving the hash value and encrypted hash address, the physician needs to be encrypted using the hash value and encoded hash address with a random number, hash the health information and index with the SHA256 hash function, and broadcast the hash value and encoded hash address to the blockchain. Furthermore, the system offers targeted defense against relevant keyword attacks. Medical data is not directly stored on IPFS, and electronic health data is encrypted before being stored on IPFS. It also takes time for the IPFS value to be encrypted before even being kept on the blockchain.

Most of the previous works lack a keyword search functionality for quickly locating relevant information. They do not mention how to retrieve the metadata from the blockchain. It is not possible to retrieve data from IPFS without the hash value of the file. Table 1 compares our proposed system with existing approaches.

**Table 1.** Comparison of existing approaches with the proposed scheme.

Constraints	Hao, J. et al. [13]	Rajalakshmi A. [14]	Sun, J. et al. [18]	Our Proposed Scheme
Delay	High delay Collected data is not directly written to an IPFS	Low delay	High delay Encryption of medical data	Low delay in uploading files to IPFS and file hash is automatically stored on BC with help of Smart contract
Tampering on the stored data	Possibilities of data tampering	No tampering	No tampering	No tampering of data as data is stored on IPFS and hash on Blockchain
Storage capacity	Less Storage capacity Stored on data server	More storage capacity	More storage capacity as the data stored on IPFS	More storage capacity as the data stored on IPFS
Heterogeneous data	Uploading only video and images on IPFS	Uploading only PDF's	Only electronic medical record	Heterogeneous data upload
Keyword Search function	No Keyword search function	No Keyword search function	No Keyword search function	Supports Keyword search function

### 3. Preliminaries

#### 3.1. IPFS

The interplanetary file system is a distributed file system protocol developed by Joan Bennett in 2015 and managed by Protocol Labs. The IPFS network consists of computers running the IPFS client software. Anyone can join the IPFS network, either as an IPFS node running the IPFS client or as a network user storing and retrieving files. Any type of file can be stored, including text, music, video, and images, which is especially useful for non-fungible tokens (NFTs). In contrast to HTTP, data in IPFS is identified by content rather than location. When we upload a file to IPFS, a hash of the content is generated. This hash identifies the content uniquely and can be used to retrieve the file. If we upload a different file, the hash will be completely different, but we can always recompute the file's hash locally to ensure it matches the original IPFS hash. We selected the IPFS protocol in our proposed scheme because it is a well-known and working decentralized file storage protocol.

#### 3.2. Ethereum

Ethereum [19] is, at its core, a decentralized global software platform that utilizes blockchain technology. It is most well-known for its native cryptocurrency, ether, abbreviated as ETH. Anyone can use Ethereum to start creating any protected digital technology. It has a token intended to be utilized by the blockchain network, but it may also be employed to pay participants for blockchain work. It is a platform for various DApps that can be deployed through smart contracts. An Ethereum Private Network is a blockchain that is completely separate from the main Ethereum network. The Ethereum Private Network is primarily used by organizations to limit blockchain read permissions.

#### 3.3. Web3.js

Web3.js [20] is a set of libraries that allows developers to communicate with a remote or local Ethereum node via HTTP, IPC, or WebSocket. You can use this library to create websites or clients that communicate with the blockchain.

#### 3.4. Ethers.js

Ethers.js [21] connects to Ethereum nodes using Alchemy, JSON-RPC, Etherscan, Infura, Metamask, or Cloudflare. Developers can use ethers.js to take advantage of full functionality for their various Ethereum needs.

### 3.5. Smart Contract

Smart contracts are programs that are implemented and stored on a blockchain when certain requirements are fulfilled. They are frequently used to automate agreement execution so that all groups have instant surety of the results even without the involvement of an additional party. They also can automate a workflow by automatically performing the next action if certain requirements are fulfilled.

### 3.6. Smart Contract Events

When a transaction is mined, smart contracts could also emit events and logs to the blockchain, which the front end can then process. Events are essential on any blockchain because they make connections between smart contracts, which are self-executing software programs that have the terms of the buyer's and seller's agreement straight integrated into lines of code for response with user interfaces. To use a smart contract, a user must first manually sign a transaction and interact with the blockchain. This is where automation can help users by simplifying things. Event-driven automation initiates processes without requiring human intervention. An automation tool can start a predefined process or workflow of smart contracts after detecting an event.

### 3.7. Decentralized Applications (DApp)

A decentralized application [22] is an application that can run autonomously, typically using smart contracts and running on a decentralized computing, blockchain, or other distributed ledger system. DApps, like traditional applications, provide some function or utility to their users.

### 3.8. React.js

React.js [23], also known as simply React, is a free and open-source JavaScript library. It is best to create user interfaces by combining code sections (components) into complete websites. We can use React as much or as little as we want. React enables developers to use separate software components across the client and server sides, which also speeds up development.

### 3.9. Dependencies

#### 3.9.1. Node Package Manager (NPM)

The node package manager (NPM) is a command-line tool for installing, updating, and removing Node.js packages from our application. It also serves as a repository for open-source Node.js packages. A package manager is essentially a set of software tools that can be used by a developer to automate and standardize package management.

#### 3.9.2. Node.js

Node.js is a simple programming language that can be used for prototyping and agile development, as well as to create extremely fast and scalable services.

#### 3.9.3. MetaMask

MetaMask is a non-custodial Ethereum-based decentralized wallet that also lets users save, buy, send, transform, and swap crypto tokens, as well as sign transactions. Using Metamask in conjunction with Web3.js in a web interface simplifies communication with the Ethereum network.

#### 3.9.4. Truffle Framework

Truffle is a set of tools that allows us to create smart contracts, write tests against them, and deploy them to blockchains. It also provides a development console and allows us to create client-side applications within our project. Truffle is the most widely used framework for creating smart contracts. It supports Solidity and Viper as smart contract languages. Truffle has three main functions: it compiles, deploys, and tests smart contracts.



#### 4. Proposed Data Storing Scheme

Our proposed scheme divides data storage, retrieval, and searching into four steps. The system uploads a file, file hash is stored on the blockchain, monitors smart contract events, and searches for relevant information.

##### 4.1. File Uploading

The main concept of the file uploading process is depicted in Figure 2. The file is selected from the DApp (browser) (1), and when the DApp form's submit button is clicked, the uploaded file is stored on IPFS (2). The hash of the file uploaded is returned to the DApp (3); this hash is the file's location. The file's hash is saved to a smart contract (4), which is subsequently kept on the blockchain (5), and the hash and other information of the uploaded file were also listed on the DApp (6), from which we can obtain all of the files we have uploaded to IPFS.

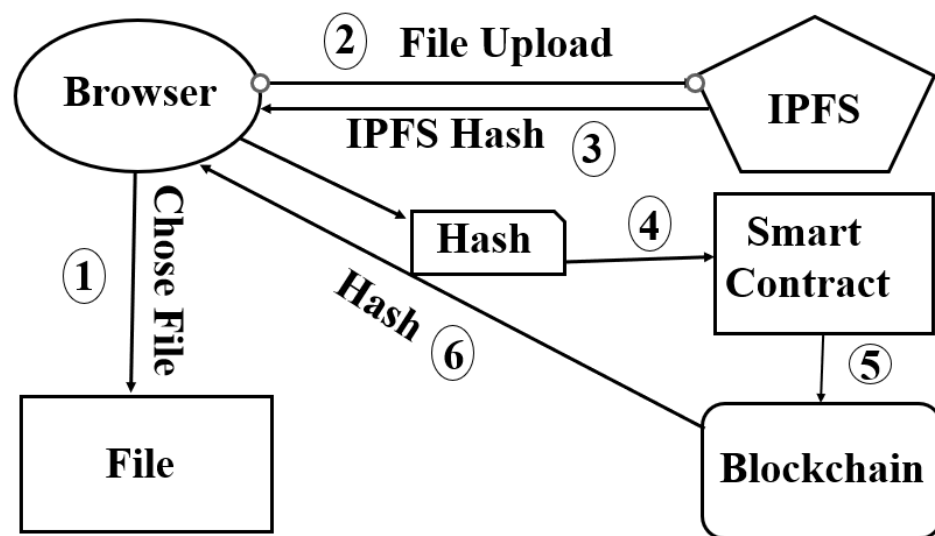


Figure 2. File Upload.

To connect to an Ethereum wallet Metamask, we used a web browser as a front end which will communicate with the blockchain and store the smart contract on it.

We will upload the file directly to an IPFS, and then IPFS will return to us a hash. We will then store this hash on the smart contract, and it will store that hash on the blockchain, allowing us to access all of the files we have created when we list them on the DApp.

A smart contract stores the hash value on the blockchain, and another smart contract lists the uploaded files on the DApp. The smart contract handles file uploading, file storage, and file listing.

Figures 3 and 4 show our smart contract. Our project's smart contract is responsible for four tasks. Define a data structure for file management, upload the files, store the file hash in the blockchain, and display the uploaded files on the DApp. We use a struct to manage the files inside Solidity. Solidity structs allow us to create more complex data types with multiple properties. By creating a struct, we can define our own type. They are useful for organizing related data. Structures can be declared outside of one contract and imported into another.

```

pragma solidity ^0.5.0; -----(1)

contract DStorage { -----(2)
    string public name = 'DStorage'; -----(3)
    uint public fileCount = 0; -----(4)
    mapping(uint => File) public files; -----(5)

    struct File { -----(6)
        uint fileId;string fileHash;
        uint fileSize;string fileType;
        string fileName; string fileDescription;
        uint uploadTime; address payable uploader;

    event FileUploaded( -----(7)
        uint fileId,string fileHash,uint fileSize,
        string fileType,
        string fileName,string fileDescription, uint uploadTime,
        address payable uploader);

```

**Figure 3.** Solidity code for creation of a blockchain register and events to facilitate interoperability (1/2).

```

constructor() public {
}

function fileUpload( -----(8)
    string memory _fileHash,uint _fileSize,
    string memory _fileType,
    string memory _fileName,
    string memory _fileDescription)

    public { -----(9)
        require(bytes(_fileHash).length > 0); // Make sure the file hash exists
        require(bytes(_fileType).length > 0); // Make sure file type exists
        require(bytes(_fileDescription).length > 0); // Make sure file description exists
        require(msg.sender!=address(0)); // Make sure uploader address exists
        require(_fileSize>0); // Make sure file size is greater than 0

        // Increment file id
        fileCount ++; -----(10)

        // Add file to the contract
        files[fileCount] = File( -----(11)
            fileCount,_fileHash,_fileSize,
            _fileType,_fileName,
            _fileDescription,now, msg.sender);

        // Trigger an event
        emit FileUploaded( -----(12)
            fileCount,_fileHash,_fileSize,
            _fileType,_fileName,_fileDescription,
            now,msg.sender);}

```

**Figure 4.** Solidity code for creation of a blockchain register and events to facilitate interoperability (2/2).

The following steps show the tasks of a smart contract:

(i) Define data structure for the management of files:

Figure 3 shows step one in modeling the file (6). We created a file object, and inside we defined a unit id, which will be the unique identifier for the file inside our smart contract. The string will be the hash of the file, and this will be its location on IPFS, and a description of the file, which contains the location of the file and events related to the uploaded file. The address-payable uploader is the person who uploads the file, and it is the Ethereum address of that person's wallet address as they are connected to the blockchain; it is like their username on the blockchain.

(ii) Store and list the files:

Step two is to store the file on IPFS, and step three is to list the event logs on the DApp. We used mapping inside of Solidity to store the files, as shown in Figure 3. Mapping is another data structure. It can be utilized to store data as key-value pairs, with the key being any of the built-in data types but just not reference types, as well as the value being any type. We created mapping (5) as shown in Figure 3. A mapping inside of Solidity is



just a key-value store. We can give it a key and a value. The data type of the key in our smart contract is an unsigned integer, and the return value is file struct (6), as shown in Figure 2. When we place a file with an id within this mapping, it will write and store it on the blockchain. Mapping is also going to give us the ability to list the files because mapping is public, and thus it gives us a function called “files” (5) that we can call, pass in the id, and fetch out each individual file. We can get back a file with all the data, such as the id, hash, file name, description, and uploader.

#### (iii) Upload File:

The solidity code has a function called fileUpload (8). “fileUpload” takes the following arguments: fileHash, fileSize, fileType, fileName, fileDescription. Whenever we upload a new file, we will just add a new file to the mapping. We created a new file (6) and put it inside the file’s mapping (5). We are going to store the file based on the id inside the mapping, as shown in (5). We stored the file onto the blockchain as shown in (11).

Inside the smart contract, Solidity has a global variable called “msg” or “message” that has many different attributes, one of which is the person calling the function, “message sender” is the Ethereum address of the person uploading the file. We created a video struct and saved it inside the “files mapping”, which we simply say “files”, pass in the id, and it will be equal to a new file (11).

**fileCount** (4) is a variable that stores the number of files that have been created. Whenever we create the smart contract, the counter value will be zero, but we can change this value inside the function (11) as **fileCount** anytime the function is called. We could write **fileCount ++** (10) and then pass in **fileCount** in (11). **fileCount** keeps track of all the files; it is basically our ID management system, and we save it inside the file mapping, which acts like our database.

#### (iv) Creating an Event:

The event allows us to know when the file was uploaded. We can create events from the Solidity code. We define an event called “fileUpload” and we pass in the same arguments as the struct (7); this is going to allow us to subscribe to the event whenever it is triggered from our application. We can trigger the upload event (12). We use the emit keyword, then **FileUploaded** which has the same name as the event (7) and we pass in the arguments file count, fileHash, fileSize, fileType, filename, file description, and now, msg.sender.

Next, we added some requirements to the function to make it robust. We can use Solidity’s **require** function (9). The **require** function checks that a set of parameters is true before the rest of the function executes. Table 2 shows the list of variables used in our smart contract.

**Table 2.** Smart Contract variables.

Variables	Why It Is Used
fileCount	Keeps track of how many files have been added to the current smart contract.
mapping File struct	key value store and lists the files Manage the files
event FileUploaded	Allows us to know when the file was uploaded
function fileUpload	Uploads new file
emit FileUploaded	Trigger an event

Recently, diverse types of formal methods are investigated to enhance the security of smart contracts, since the compromise of smart contracts can lead to a catastrophic monetary loss [24]. However, our smart contract codes have not been analyzed using those formal methods yet, and we will verify our codes in our future work.

Our first project element is a private Ethereum blockchain that will act as the back end for our DApp. Ethereum nodes maintain an archive of the blockchain’s code. The information is dispersed throughout the network. The Geth is utilized to run an Ethereum node.

By running a node on the Ethereum network, we could also perform transactions as well as communicate with smart contracts. The uploaded file's hash is saved in a smart contract, and then immutably stored also on the Ethereum blockchain.

The next component is IPFS, which enables us to keep files in a distributed fashion. Because files are large, storing megabytes and gigabytes of files on the blockchain may not be feasible. This is where IPFS comes into play. It has nodes, just like Ethereum, and we distribute files that cannot be tampered with across the network. IPFS uses hashes. When you upload a file to IPFS, it will be stored somewhere and identified by its hash. We run our own IPFS node, which supports an IPFS gateway for file retrieval and storage and runs the IPFS Daemon server. We cannot store or retrieve data unless the Daemon server is up and running, or unless we link to public gateways such as Infura [25].

When a user uploads CCTV footage to our DApp, they can specify the location as well as event details such as whether it was an accident or a traffic violation. This information is fed into the DApp as a file description. This information is critical when uploading a video to the DApp because users can quickly search for location and event information using the DApp's keyword search function.

We first must import and link our Ethereum blockchain account to Metamask before we can use the DApp. Our web browser now supports blockchain networks, and we can upload files to IPFS using our custom-designed DApp user interface (UI). First, we must select the file, enter its description (such as file event and location), and then click the submit button. When we click the submit button, the file is sent to IPFS and we receive the IPFS result, which contains the hash value and path of the file. Metamask directs us to accept the transaction, save the hash in a smart contract, and store the smart contract on the blockchain via a confirmation pop-up. To store the hash on the blockchain, we should pay some gas in the manner of ethers. When we confirm the Metamask transaction, the hash of the uploaded file is preserved on the Ethereum blockchain.

The DApp monitors the "file upload" event and updates the DApp's User interface automatically. The event log of the smart contract is generated by retrieving and displaying all events from the smart contract within our DApp. The smart contract event log includes the file no, file description, type of file, file size, timestamp, Ethereum information of the uploaded person, and the hash value of the file after it has been stored in IPFS. By clicking on the file description, individuals may view the uploaded files in their web browser. The hash value does not need to be remembered or stored separately by the user.

#### 4.2. Keyword Searching

Users of the blockchain network can view transaction details but cannot identify the individuals who made the transactions. On our DApp, we can see the transactions and use the data for keyword searching.

##### (i) Read information from the blockchain:

When events occur in the smart contract, the smart contract emits events in order to communicate with DApps and other smart contracts. When we invoke a smart contract function, it has the ability to generate an event. It is critical for us to be able to listen to these events in real time when developing DApps.

To listen for smart contract events, we used Ethers.js smart contract event listener. To communicate with a smart contract using Ethers.js, we must first create a new contract object with Ethers as shown in step (1) of Figure 5.

```

Creating Instances
new ethers.Contract( address , abi , signerOrProvider ) ----- (1)

const abi = [] -----(2)
const address = "" -----(3)
const provider = new ethers.providers.JsonRpcProvider('HTTP://localhost:8545'); -----(4)
const contract = new ethers.Contract(address, abi, provider); -----(5)

contract.queryFilter( 'event' ) ----- (6)
contract.queryFilter( event [ , fromBlockOrBlockHash [ , toBlock ] ) = Promise< Array< Event > > ----- (7)
|Return Events that match the event.

```

**Figure 5.** Ethers.js filter to read events from blockchain.

As shown in steps (2), (3), and (4) of Figure 5, we need the blockchain address for the smart contract, the ABI of the smart contract, and the signer or provider (4). The ABI is a JSON object that describes how the smart contract works; it describes the interface, which essentially means what functions the smart contract has, what function arguments it accepts, and what it responds to when we try to read data from it. Ether.js allows us to store ABIs as an array and only pull in the parts we want when we are setting up a smart contract object. We require file upload information for our project, so we included ABI, which is related to the file upload event. Then, we need a provider or a signer; in our project, we have a provider. A provider is an abstraction of an Ethereum network connection that provides a concise, consistent interface to standard Ethereum node functionality. We take our smart contract ABI and create a new contract address ABI, and then we provide all of the required information as shown in step (5) of Figure 5.

We used `contract.queryFilter` to filter the information, as shown in step (6) of Figure 5. Using this command, we will examine every single `FileUploaded` event that has ever occurred on our blockchain. We include this filter to reduce the search space inside the Ethereum blockchain. Ethers.js allows us to examine the `FileUploaded` events and specify which blocks we want to examine as shown in step (7) of Figure 5.

(ii) Keyword search text file creation:

We can create a text file for keyword searches once the events are retrieved from the blockchain. The smart contract events are written into a text file. We store only necessary information in the text file, e.g., information such as file name, event type, location, Ethereum account number, and smart contract.

Figure 6 shows how to retrieve data from the blockchain and conduct keyword searches. To listen to smart contract events, we used a command prompt to send requests to the blockchain (1). Blockchain responded with a filtered smart contract event log containing all of the information about the uploaded file, including the smart contract address, file name, file hash and description, Ethereum address of the uploaded file, and so on (2). When we received a smart contract event log, we saved some of the event logs in a text file (3). We wrote code in `react.js` to filter the results and search for keywords on the DApp. When a user searches for a keyword on the DApp, the request is sent to a text file containing smart contract events, which is then filtered, and the result is returned to the DApp (4). Users can look up a word or an alphabet.

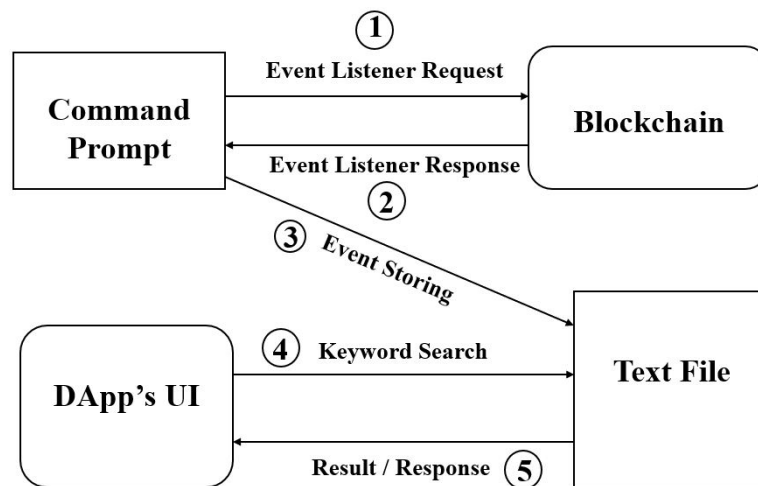


Figure 6. Keyword Search Function of the proposed DApp.

## 5. Implementation

On the Windows 10 operating system, we used a private Ethereum blockchain to implement a proposed scheme. The Ethereum core network is not connected to a private Ethereum network. Organizations primarily use it to limit blockchain read permissions. Installing geth/parity allows the current node to join the Ethereum network and download the blockchain to local storage. We used Go Ethereum to create our Ethereum blockchain (Geth).

### 5.1. Steps to Create Private Ethereum Network

The following steps show how we built our private Ethereum network:

#### 5.1.1. Download “Geth”

Go Ethereum (Geth) can be directly downloaded and installed from [geth.ethereum.org](https://geth.ethereum.org), accessed on 16 February 2023. Because Geth is a command line interface, we execute all commands from the command line. After installing Geth on our system, we typed geth and pressed enter in a command prompt and obtained the output as shown in Figure 7.

```

PS C:\Ethereum\privatechain\eth> geth
INFO [09-22|13:40:54.475] Starting Geth on Ethereum mainnet...
INFO [09-22|13:40:54.477] Bumping default cache on mainnet
provided=1024 updated=4096
INFO [09-22|13:40:54.479] Maximum peer count
ETH=50 LES=0 total=50
WARN [09-22|13:40:54.505] Sanitizing cache to Go's GC limits
provided=4096 updated=2669
INFO [09-22|13:40:54.506] Set global gas cap
cap=50,000,000
INFO [09-22|13:40:54.506] Allocated trie memory caches
clean=400.00MiB dirty=667.00MiB
INFO [09-22|13:40:54.508] Allocated cache and file handles
database=C:\Users\IT\AppData\Local\Ethereum\geth\chaindata cache=1.30GiB handles=8192
INFO [09-22|13:40:54.545] Opened ancient database
database=C:\Users\IT\AppData\Local\Ethereum\geth\chaindata\ancient
readonly=false
INFO [09-22|13:40:54.546] Writing default main-net genesis block
INFO [09-22|13:40:54.700] Persisted trie from memory database
nodes=12356 size=1.78MiB time=31.8892ms gcnodes=0 gcsz=0.008 geti
me=0s livenodes=1 liveness=0.008
  
```

Figure 7. Geth command.

We used the geth command to connect to a blockchain, and the geth command will run in fast sync mode. Fast sync is Geth’s current default sync mode. Fast Sync nodes download the headers of each block and retrieve all the nodes beneath them until they

reach the leaves. Instead of reprocessing all transactions that have ever taken place, fast sync downloads the blocks but only validates the affiliated proof-of-works (which could take weeks). When we stop and restart the geth, it will operate in full sync mode. Full sync needs to download all blocks and incrementally generate the blockchain state by running each block since genesis. The data size of the Ethereum blockchain is currently around 800–1000 gigabytes, and we do not need to download the entire Ethereum blockchain on our system.

#### 5.1.2. Make a Folder for Our Private Ethereum Network

For the private Ethereum network, we created a separate folder called “Private Ethereum”. This folder separates the Ethereum private network files from the public files.

#### 5.1.3. Construct a Genesis Block

In blockchain, all transactions are recorded in the form of blocks in sequential order. There are an infinite number of blocks, but there is always one distinct block that gives rise to the entire chain, known as the genesis.

The genesis block, also known as Block 0 or Block 1, is the first block ever recorded on its respective blockchain network. There are no transactions. The genesis block is used to initialize the blockchain, as shown in Figure 8. A genesis block is required to create a private blockchain. The genesis block can be created with any text editor and saved with the JSON extension in the **Private Ethereum** folder. Figure 9 shows the genesis block file.

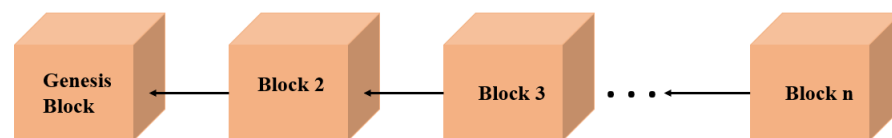


Figure 8. Genesis block in a blockchain.

```

{} genesis.json x
privatechain > {} genesis.json > ...
1  {
2    "config": {
3      "chainId": 2022,
4      "homesteadBlock": 0,
5      "eip150Block": 0,
6      "eip155Block": 0,
7      "eip158Block": 0,
8      "byzantiumBlock": 0,
9      "constantinopleBlock": 0,
10     "petersburgBlock": 0,
11     "ethash": {}
12   },
13   "difficulty": "0x20000",
14   "gasLimit": "0x9000000",
15   "alloc": {}
16 }
  
```

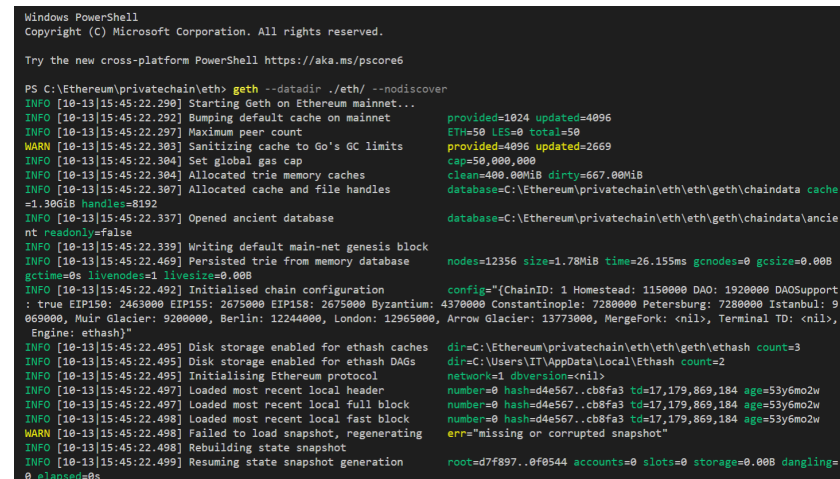
Figure 9. Genesis block file.

#### 5.1.4. Run the Genesis File

To extract the genesis file, we open the **Private Ethereum** folder in Visual Studio Code and run the command **geth init ./genesis.json -datadir eth**. Eth is the name of a folder. Geth is connected to the genesis file after running the above command.

### 5.1.5. Set Up the Private Network

We created a private network in which multiple nodes can add new blocks. We must use the command **geth --datadir ./eth/ --nodiscover** to accomplish this. When **--nodiscover** is used to start a geth node, it prevents the node from being discovered by the network's bootnode. Every time the private network chain is needed, commands in the console must be executed to connect to the genesis file and the private network. A private Ethereum network and a personal blockchain are now available. Figure 10 shows the running status of a private Ethereum network.



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

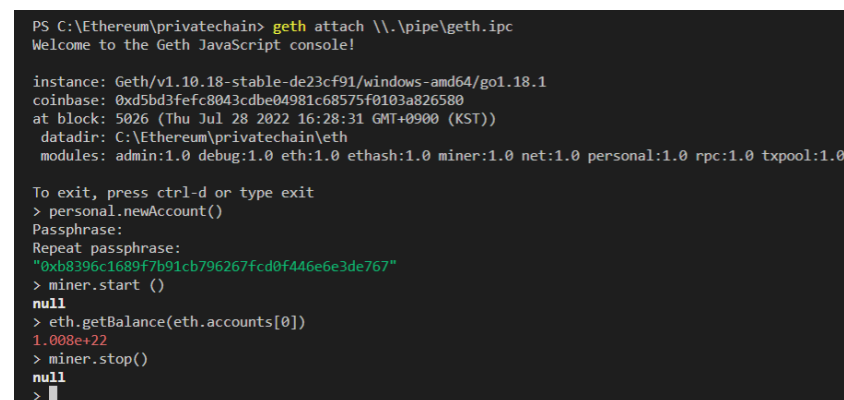
Try the new cross-platform PowerShell https://aka.ms/powershell

PS C:\Ethereum\privatechain\eth> geth --datadir ./eth/ --nodiscover
INFO [10-13|15:45:22.290] Starting Geth on Ethereum mainnet... provided=1824 updated=4096
INFO [10-13|15:45:22.292] Rumping default cache on mainnet ETH=50 LRU=0 total=50
INFO [10-13|15:45:22.297] Maximum peer count provided=4096 updated=2669
WARN [10-13|15:45:22.303] Sanitizing cache to Go's GC limits cap=50,000,000
INFO [10-13|15:45:22.304] Set global gas cap clean=400.00MiB dirty=667.00MiB
INFO [10-13|15:45:22.304] Allocated trie memory caches database=C:\Ethereum\privatechain\eth\eth\geth\chaindata\cache
INFO [10-13|15:45:22.307] Allocated cache and file handles database=C:\Ethereum\privatechain\eth\eth\geth\chaindata\ancient
INFO [10-13|15:45:22.337] Opened ancient database nt readonly=false
INFO [10-13|15:45:22.339] Writing default main-net genesis block nodes=12356 size=1.78MiB time=26.155ms gcnodes=0 gcsizes=0.008
INFO [10-13|15:45:22.469] Persisted trie from memory database gctime=0s livenodes=1 liveness=0.008
INFO [10-13|15:45:22.492] Initialised chain configuration config="{ChainID: 1 Homestead: 1150000 DAO: 1920000 DAOsupport: 4370000 Constantinople: 7280000 Petersburg: 7280000 Istanbul: 9069000 Muir Glacier: 9200000 Berlin: 12244000 London: 12965000 Arrow Glacier: 13773000 MergeFork: <nil>, Terminal TD: <nil>, Engine: ethash}"
INFO [10-13|15:45:22.495] Disk storage enabled for ethash caches dir=C:\Ethereum\privatechain\eth\eth\geth\ethash count=3
INFO [10-13|15:45:22.495] Disk storage enabled for ethash DAGs dir=C:\Users\IT\AppData\Local\Ethash count=2
INFO [10-13|15:45:22.495] Initialising Ethereum protocol network=1 dbversion=<nil>
INFO [10-13|15:45:22.497] Loaded most recent local header number=0 hash=d4e567..cb8fa3 td=17,179,869,184 age=53y6mo2w
INFO [10-13|15:45:22.497] Loaded most recent local full block number=0 hash=d4e567..cb8fa3 td=17,179,869,184 age=53y6mo2w
INFO [10-13|15:45:22.498] Loaded most recent local fast block number=0 hash=d4e567..cb8fa3 td=17,179,869,184 age=53y6mo2w
WARN [10-13|15:45:22.498] Failed to load snapshot, regenerating err="missing or corrupted snapshot"
INFO [10-13|15:45:22.498] Rebuilding state snapshot root=d7f897..0f0544 accounts=0 slots=0 storage=0.008 dangling=
INFO [10-13|15:45:22.499] Resuming state snapshot generation
0 elapsed=0s
```

Figure 10. Private Ethereum network.

### 5.1.6. Make Externally Owned Account (EOA)

EOAs are controlled by users who have access to the account's private keys. These accounts, which can both send transactions and trigger contract accounts, are typically used in conjunction with a wallet. To manage the blockchain network, EOA is required. To make it, we launched Geth in two windows. One terminal to run Geth as shown in Figure 10 and another terminal to create EOA. We entered the command **geth attach \\.\\pipe\\geth.ipc** in the second terminal (console window). This will connect the second terminal to the private Ethereum network established in Figure 10. We used the command **personal.newAccount()** to create a new account. After executing this command, we entered our password to obtain our account number and saved it for future use as shown in Figure 11.



```
PS C:\Ethereum\privatechain> geth attach \\.\\pipe\\geth.ipc
Welcome to the Geth JavaScript console!

instance: Geth/v1.10.18-stable-de23cf91/windows-amd64/go1.18.1
coinbase: 0xd5bd3f8c8043cdbe04981c68575f0103a826580
at block: 5026 (Thu Jul 28 2022 16:28:31 GMT+0900 (KST))
datadir: C:\Ethereum\privatechain\eth
modules: admin:1.0 debug:1.0 eth:1.0 ethash:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 txpool:1.0

To exit, press ctrl-d or type exit
> personal.newAccount()
Passphrase:
Repeat passphrase:
"0xb8396c1689f7b91cb796267fcd0f446e6e3de767"
> miner.start()
null
> eth.getBalance(eth.accounts[0])
1.008e+22
> miner.stop()
null
>
```

Figure 11. Externally owned account, Mining Start and Stop.

### 5.1.7. Ethereum Mining on Our Private Chain

If we mine on the Ethereum main chain, we will need expensive equipment with powerful graphics processors. ASICs are typically used for this but high performance is not



required in our private network, and we can begin mining with the command **miner.start ()** as shown in Figure 11.

After a few seconds, some ether was found in the default account if the balance status is checked as shown in Figure 11. To check the balance, we used the command **eth.getBalance(eth.accounts[0])**. Figure 12 shows the mining process. We used the command **miner.stop()** to stop mining as shown in Figure 11.

```

minial Help genesis.json - Ethereum - Visual Studio Code
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
WARN [10-13|16:13:25.647] Please remember your password!
INFO [10-13|16:13:37.223] Updated mining threads threads=12
INFO [10-13|16:13:37.224] Transaction pool price threshold updated price=1,000,000,000
INFO [10-13|16:13:37.227] Commit new sealing work number=2303 sealhash=2852b8..966f85 uncles=0 t
xs=0 gas=0 fees=0 elapsed=3.588ms
INFO [10-13|16:13:37.229] Commit new sealing work number=2303 sealhash=2852b8..966f85 uncles=0 t
xs=0 gas=0 fees=0 elapsed=5.136ms
INFO [10-13|16:13:50.453] Successfully sealed new block number=2303 sealhash=2852b8..966f85 hash=43052
1..0af4e7 elapsed=13.226s
INFO [10-13|16:13:50.455] ^ mined potential block number=2303 hash=430521..0af4e7
INFO [10-13|16:13:50.456] Commit new sealing work number=2304 sealhash=9ebc9b..7a3a7d uncles=0 t
xs=0 gas=0 fees=0 elapsed=2.995ms
INFO [10-13|16:13:50.553] Commit new sealing work number=2304 sealhash=9ebc9b..7a3a7d uncles=0 t
xs=0 gas=0 fees=0 elapsed=100.740ms
INFO [10-13|16:13:51.548] Successfully sealed new block number=2304 sealhash=9ebc9b..7a3a7d hash=d8e02
a..bab826 elapsed=1.094s
INFO [10-13|16:13:51.592] ^ mined potential block number=2304 hash=d8e02a..bab826
INFO [10-13|16:13:51.613] Commit new sealing work number=2305 sealhash=aafa2e..d8ae29 uncles=0 t
xs=0 gas=0 fees=0 elapsed=63.829ms
INFO [10-13|16:13:51.683] Commit new sealing work number=2305 sealhash=aafa2e..d8ae29 uncles=0 t
xs=0 gas=0 fees=0 elapsed=133.642ms
INFO [10-13|16:13:59.519] Successfully sealed new block number=2305 sealhash=aafa2e..d8ae29 hash=6d0e9
7..751894 elapsed=7.909s
INFO [10-13|16:13:59.522] ^ mined potential block number=2305 hash=6d0e97..751894
INFO [10-13|16:13:59.523] Commit new sealing work number=2306 sealhash=59f693..33b263 uncles=0 t

```

Figure 12. Mining Process.

#### 5.1.8. Connecting the Private Ethereum Network to Metamask

We closed the terminal in which our private network was running and opened a new terminal and typed the command **geth --datadir ./eth/ --nodiscover --http --http.addr "localhost" --http.port "8545" --http.corsdomain="\*" --http.api web3,eth,debug,personal,net --ws.api web3,eth,debug,personal,net --networkid 7777 --allow-insecure-unlock**, as shown in the Figure 13 and now our private Ethereum is connected to Metamask.

Explanation of the used commands as follows:

- **http.addr** value:  
Listening interface for HTTP-RPC servers (default: "localhost").
- **http.port** value:  
Listening port for HTTP-RPC server (default: 8545).
- **http.corsdomain** value:  
A list of domains separated by commas that will accept cross-origin queries (browser enforced). Because the HTTP server can be accessed from any local application, the server includes additional safeguards to prevent API abuse from web pages. The server must be configured to accept Cross-Origin requests in order to allow API access from a web page. The **—http.corsdomain** flag is used to accomplish this. The **—http.corsdomain** command accepts wildcards, allowing access to the RPC from any location: **—corsdomain '\*'**.
- **http.api** value:  
APIs accessible via the HTTP-RPC protocol.
- **ws.api** value: APIs accessible via the WS-RPC interface.
- **nodiscover**:  
The peer discovery mechanism is disabled.
- **networkid** value:  
Sets network id explicitly.
- **allow-insecure-unlock**:  
When account-related RPCs are exposed via http, this allows for insecure account unlocking.

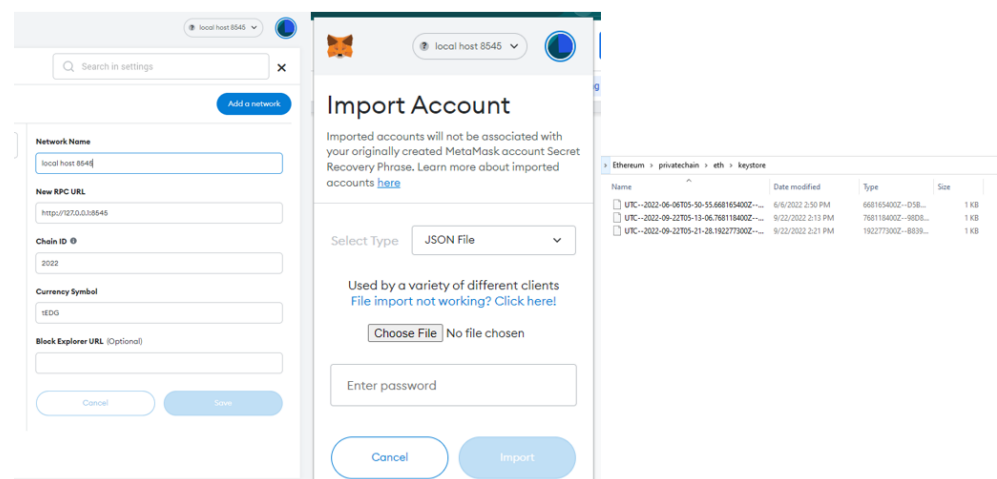


Figure 13. Importing Ethereum account in Metamask.

We launched Metamask and added the Network “Local Host 8545” with the Chain ID “2022”. It is the chain ID we specified in our private Ethereum network’s genesis block. By importing a JSON file from our private Ethereum folder, we imported a private Ethereum account. The JSON file can be found in the keystore’s Private Network folder. Figure 13 depicts how to add a Private Ethereum account to Metamask.

### 5.2. Running Our Own IPFS Node

To store information on IPFS, we must run an IPFS Daemon server on our own IPFS node. To use IPFS, we must first download and install the Go language from the golang website, then go to the IPFS command line install page and download “install go-ipfs”. We navigate to the download path, extract the files to C drive, and then run ipfs.exe to start the Daemon server, as shown in Figure 14.

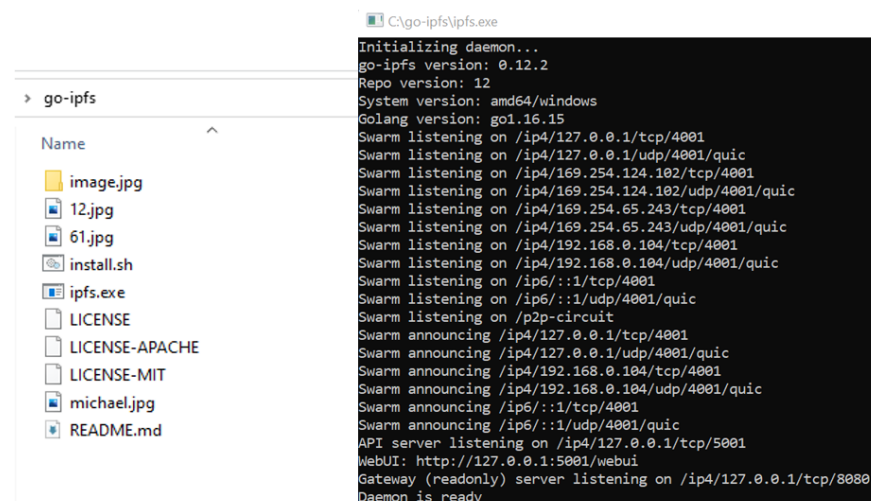


Figure 14. IPFS Execution and Daemon server.

### 5.3. Deploying Smart Contract

A smart contract stores the hash of the uploaded file. To make smart contracts in the Solidity programming language, the Truffle framework is used. The Truffle Suite is a collection of tools specifically designed for Ethereum blockchain development. The suite includes three pieces of software. Truffle is capable of helping compile and deploy smart contracts in addition to injecting them into web apps and building DApp front ends. Truffle is now a popular Ethereum Blockchain IDE.

#### 5.4. File Uploading and Retrieving

After writing the smart contract, deploying, and publishing it to our Ethereum blockchain, we then utilize Metamask to connect our DApp to the Ethereum blockchain. A Metamask is required to communicate with the blockchain. The client-side application, which is also going to communicate with IPFS, was built with React.

Figures 15 and 16 show how we initially deployed the smart contract to Ethereum, then launched the DApp with the command **npm run start**, imported an Ethereum account into Metamask, and linked Metamask to our DApp. Figures 17 and 18 show how to submit a file to IPFS, deposit the file's hash in a smart contract, record the smart contract on the Ethereum blockchain, and successfully retrieve the file using our DApp.

```

2_deploy_contracts.js
=====
Deploying 'DStorage'
-----
> transaction hash: 0xc1078dc7784d28a47c1756642104824bf8a1b3f21b9b6ea01aa7730ac7cd10e8
> Blocks: 0
> Seconds: 0
> contract address: 0x5cAa8495318D7E9aCE6C2a8719Fb7Dc7504497Ef
> block number: 1186
> block timestamp: 1654497105
> account: 0xd5bD3FefC8043CdbE04981C68575f0103a826580
> balance: 2372
> gas used: 1047485 (0xffbbd)
> gas price: 20 gwei
> value sent: 0 ETH
> total cost: 0.0209497 ETH

> Saving migration to chain.
> Saving artifacts
-----
> Total cost: 0.0209497 ETH

Summary
=====
> Total deployments: 2
> Final cost: 0.02622452 ETH

```

Figure 15. Smart contract deploy.

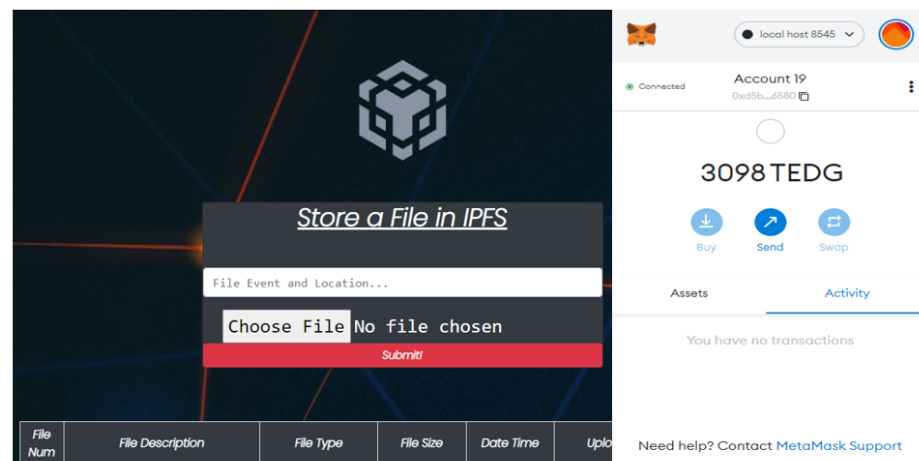


Figure 16. Connecting Metamask to the DApp.

We chose the file and entered the location as well as the location of the file in the user interface of DApp after logging into Metamask, then clicked the submit button also confirmed the transaction of Metamask, as shown in Figure 17. To deploy the smart contract, upload files, and store hash values on the blockchain, we start and maintain mining.

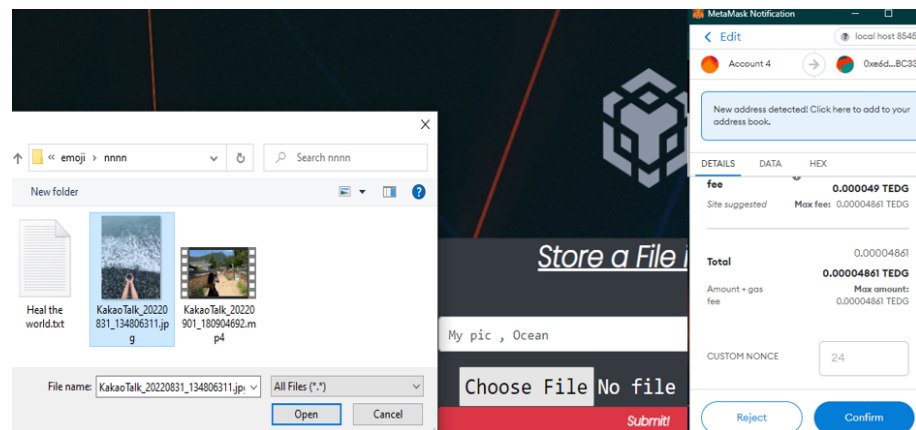


Figure 17. Choosing a file and confirming Metamask transaction.

As the transaction is confirmed, the DApp listens for the event “File Upload” and updates the DApp’s user interface automatically. Whenever a transaction has been mined, smart contracts generate events and logs to the blockchain, which can then be processed by the front end. Our DApp retrieves and displays all smart contract events. It is referred to as a “smart contract event log”. The event log of the smart contract contains the file number, file description (which includes an event and location of the file), type of the file, file size, date and time, the uploader’s Ethereum account details, and the hash of the file. By having to click on the file’s file details, users are able to view the uploaded files through their web browser. Figure 18 depicts a smart contract event log and various file types retrieved.

File Num	File Name	File Type	File Size	Date Time	Uploader	File Hash
10	My Pic	Image/jpeg	898 KB	8:10 PM 1/9/2022	0xF1348D4L	QmSL2JNAJaoXw9TTmGPYGVUD5LxvB8b98dcs28NpP
9	Video	video/mp4	1 MB	6:09 PM 1/9/2022	0xF1348D4L	QmaK88EPCqk6b3wYKs7L8uS8fm2JITyababTfY1ho
8	Heal the w	text/plain	80 Bytes	5:50 PM 1/9/2022	0xF1348D4L	QmPoCqIngU2kUnogFwMLBdStDcZTazHBQw56yWwqGy

Figure 18. Event log and file retrieve.

### 5.5. Keyword Searching

Our DApp supports the keyword search method. In order to conduct keyword searches, we obtain event information from Blockchain. Smart contracts could even emit logs as well as events to the blockchain whenever an Ethereum transaction is mined, which the front end can then process. An event broadcasts information about a file upload, and we could have access to all of the events so that we could listen to them in real time, or we could just use them to obtain all of the most recent file uploads on the blockchain.

We can read smart contract events outside of the DApp’s user interface by using Web3.js or Ethers.js. In our implementation, Ethers.js is used to read smart contract events. We only have one event in our smart contract, so we use a filter to retrieve information from that event, which is File Upload. A smart contract event log is shown in Figure 19.

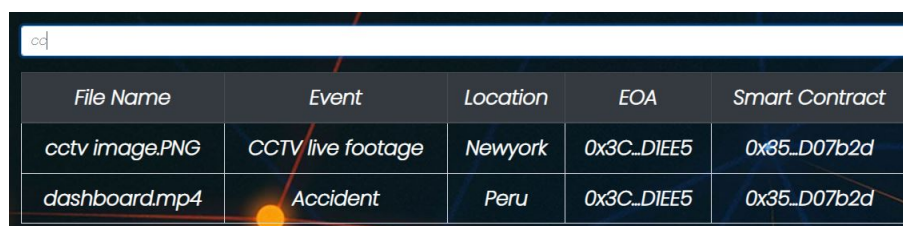
```

[BigNumber],
'QmPr1saZTDteYFViYA7MRD2M7174XZ7QokW1J67NGnGZ7C',
[BigNumber],
'video/mp4',
'dashboard.mp4',
'Accident, Peru',
[BigNumber],
'0x3Cf720BC071dFF9ffc8879b7E4C514343c0D1EE5',
fileId: [BigNumber],
fileHash: 'QmPr1saZTDteYFViYA7MRD2M7174XZ7QokW1J67NGnGZ7C',
fileSize: [BigNumber],
fileType: 'video/mp4',
fileName: 'dashboard.mp4',
fileDescription: 'Accident, Peru',
uploadTime: [BigNumber],
uploader: '0x3Cf720BC071dFF9ffc8879b7E4C514343c0D1EE5'

```

Figure 19. Smart contract events.

The smart contract events are then written to a text file, allowing our DApp to conduct keyword searches. We store only necessary information in the text file, e.g., information such as file name, event type, location, Ethereum account number, and smart contract. When looking for sensitive information on the DApp, keyword searching is essential. Entering an alphabet or a keyword into keyword searching will filter the results to show only the keyword we entered. In the case of an alphabet search, the DApp will display all events that include the letter we typed into the search box. This method makes navigating an event easier and more efficient. Keyword searching is shown in Figure 20.



File Name	Event	Location	EOA	Smart Contract
cctv image.PNG	CCTV live footage	Newyork	0x3C_D1EE5	0x85_D07b2d
dashboard.mp4	Accident	Peru	0x3C_D1EE5	0x35_D07b2d

Figure 20. Keyword Searching.

## 6. Performance Evaluation

The majority of applications we use today are centralized, which means they are managed by a single authority. Google [26] and Facebook [27], for example, retain complete ownership of their respective products, running their apps and storing user data on private servers and databases. While this gives Google and Facebook control over their applications and user experiences, it can also be discouraging to users. Users of centralized apps have little control over their data or experience within the app. They must have faith in the app's developer to listen to their feedback, provide product services, and treat them and their data with dignity. However, with other centralized applications facing backlash over privacy and the monetization of user data, many users are wary of relying on them.

Centralized applications run programs and store critical user information on centralized servers. The entire application may fail if a single, central server is compromised. DApps enable users to complete transactions, verify claims, and collaborate in real time without relying on a centralized intermediary.

Our DApp operates on a peer-to-peer network, similar to a distributed ledger, with each network member contributing to the program. Each of the roles that a central server would normally provide, from computing power to storage, is distributed across the network. We do not need to keep and secure a central server, and users can directly participate in the app's operation. Our system is robust to system failure. There is no single point of failure in our DApp and is distributed across a network of public nodes, with copies of critical information distributed



among them. The application is unaffected if one or more IPFS nodes are compromised. Even if there is a virus attack, a hardware failure, or the system is turned off, the user can still retrieve the uploaded files and perform keyword searches.

When a user uploads data to IPFS, it is chopped into smaller chunks, hashed, and assigned a unique content identifier (CID), which serves as a fingerprint. This makes it faster and easier to store small amounts of data on the network. A cryptographic hash (CID) is generated for each piece of data, making each upload to the network unique and resistant to security breaches or tampering.

The experiment we conducted demonstrates that our DApp is resistant to system failure, robust, and transparent.

The experiments we carried out are listed below.

#### Scenario 1:

In Scenario 1, the system unexpectedly shuts down, and when it is restarted, the DApp's event log vanishes, as illustrated in Figure 21. We can retrieve the event log outside of the DApp using smart contract event listeners. In Figure 19, we used Ethers.js to retrieve the event log. The data associated with the uploaded file is included in the event log. As a result, system failure has no effect on the uploaded data.

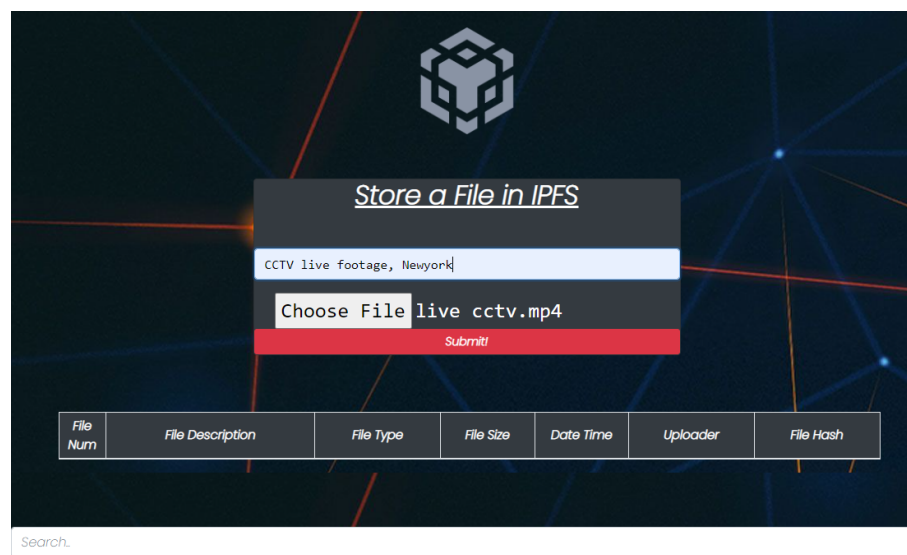


Figure 21. No event log listed on the DApp.

#### Scenario 2:

The information in the keyword search text file was accidentally deleted in Scenario 2 as shown in Figure 22, and we were unable to perform the keyword search on the DApp. As demonstrated in Scenario 1, we recreated the keyword search text file using information retrieved from the smart contract event log and performed a keyword search as illustrated in Figure 23. Table 3 summarizes the scenarios of performance evaluation.

Table 3. Performance Evaluation Scenario Summarization.

Scenario #	Description
1	The system unexpectedly shuts down. When the system restarted, the DApp's event log vanished. We used ether.js to retrieve the event log.
2	The information in the keyword search text file was accidentally deleted. We recreated the keyword search text file by using information retrieved from the smart contract event log and performed a keyword search.



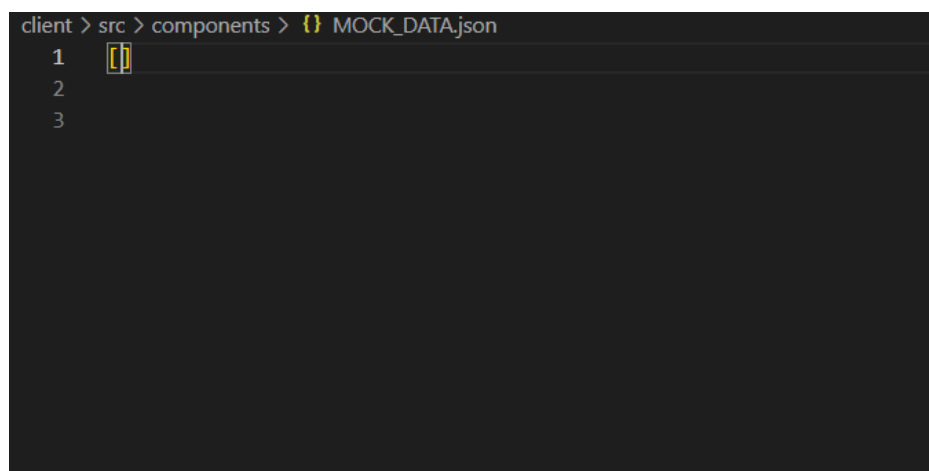


Figure 22. Text file with no data.

File Num	File Description	File Type	File Size	Date Time	Uploader	File Hash
4	You rock my world I	audio/mp3	8 MB	8:58 PM 25/8/2022	0x937bdc31	0m1f9q0eac2u3gulaC3eCp00002540n8b8x7WgHgn
3	CCTV live footage, II	video/mp4	5 MB	8:57 PM 25/8/2022	0x937bdc31	0m02bU4fVg3m0e0u9fHmV8IM0P2UJ0eHfYnIDbf
2	CCTV live footage, II	video/mp4	5 MB	8:58 PM 25/8/2022	0x937bdc31	0m02bU4fVg3m0e0u9fHmV8IM0P2UJ0eHfYnIDbf
1	CCTV live footage, II	video/mp4	5 MB	8:58 PM 25/8/2022	0x937bdc31	0m02bU4fVg3m0e0u9fHmV8IM0P2UJ0eHfYnIDbf

File Name	Event	Location	EOA	Smart Contract
Michael Jackson - You Rock My World	You rock my world	Earth	0x93_02483	0xCF02_0568D
cctv image.PNG	CCTV live footage	Newyork	0x3C_DIEE5	0x35_D07b2d
dashboard.mp4	Accident	Peru	0x3C_DIEE5	0x35_D07b2d
Michael Jackson - You Rock My World	Music Video	Earth	0x3C_DIEE5	0x35_D07b2d

Figure 23. Keyword Search.

If a malicious actor manages to compromise the blockchain network, any changes are visible on a public network, allowing both users and developers to respond quickly. Our DApp operates on a public ledger, which means that anyone with internet access can participate in the application and network. As a result, anyone can view the transaction record and any changes made to those records. Therefore, this system can provide better transparency than centralized applications can provide. On a publicly distributed ledger, no central entity can revoke transparency, limit viewership, or censor participation.

## 7. Conclusions

In this paper, we present the design and implementation of a decentralized application that uses Ethereum blockchain and IPFS to store CCTV and black box footage securely and efficiently. The DApp allows users to easily manage their storage. For scalability, only hashes of the files are stored on the blockchain via smart contracts. Our proposed scheme works in a decentralized manner. When a file is uploaded, the DApp listens for the event File Upload and automatically updates the DApp's user interface. All smart contract events are fetched and displayed on our DApp. The extracted information is called a smart contract event log, and it includes information about the file, timestamp, the uploader's account information, and the hash of the IPFS file returned. By clicking on the file's description, users can gain access to it. The selected file is then displayed in the web browser. DApp also includes a keyword search

feature to help us find any information quickly. To filter and read data from the blockchain, we used ether.js' smart contract event listener and contract.queryFilter. We used the smart contract address as well as the smart contract's ABI. The smart contract events are then written into a text file. The text file only contained necessary information, such as the file name, event type, location, Ethereum account number, and smart contract. Our experiment shows that our DApp is not affected by system failure. We can secure an application by managing the data in a decentralized manner. Because our DApp runs on a public ledger, anyone with internet access can participate in the application and network. As a result, anyone can view and modify the transaction record. As a result, unlike centrally managed applications, this system provides greater transparency. We anticipate that our DApp can be used in a variety of fields, such as for keeping records of student research securely at universities, the medical information of patients at hospitals, and customer information at banks due to its ability to store various file types.

In our current system, the access control function is not included in the smart contract yet, and thus, the hash values of one's files can be exposed to anyone who knows his or her smart contract address. We will investigate the access control scheme for the smart contract to resolve this issue in our future work. In addition, we will also verify the source code of our smart contract using well-known formal methods.

Recently, Ethereum has been upgraded by changing its consensus mechanism from proof-of-work (PoW) to proof of stake (PoS), and this new version is also known as Ethereum 2.0. However, this new consensus mechanism has not been verified intensively compared to the PoW mechanism, and thus, we used an old version of Ethereum and its corresponding Ethereum Virtual Machine (EVM) environment in this paper. We will implement and investigate our proposed system on the new version of Ethereum in our future work.

**Author Contributions:** Conceptualization, N.S. and S.Y.N.; data curation, N.S.; formal analysis, N.S.; methodology, N.S.; project administration, N.S. and S.Y.N.; resources, N.S.; software, N.S.; supervision, S.Y.N.; validation, N.S.; visualization, N.S.; writing—original draft, N.S.; writing—editing and review, N.S. and S.Y.N. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research was supported in part by the National Research Foundation of Korea (NRF), with a grant funded by the Korean government (MSIT) (2020R1A2C1010366). This research was supported in part by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2021R1A6A1A03039493).

**Data Availability Statement:** No new data were created or analyzed in this study. Data sharing is not applicable to this article.

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

## References

1. Mateen, A.; Khalid, A.; Nam, S.Y. Management of Traffic Accident Videos using IPFS and Blockchain Technology. *KICS Summer Conf.* **2022**, *1*, 1366–1368.
2. Singh, A.; Chatterjee, K. Cloud security issues and challenges: A survey. *J. Netw. Comput. Appl.* **2017**, *79*, 88–115. [CrossRef]
3. Shin, Y.; Koo, D.; Hur, J. A survey of secure data deduplication schemes for cloud storage systems. *ACM Comput. Surv.* **2017**, *49*, 1–38. [CrossRef]
4. Yinghui, Z.; Dong, Z.; Deng, R.H. Security and privacy in smart health: Efficient policy-hiding attribute-based access control. *IEEE Internet Things J.* **2018**, *5*, 2130–2145.
5. Zhang, Y.; Chen, X.; Li, J.; Wong, D.S.; Li, H.; You, I. Ensuring attribute privacy protection and fast decryption for outsourced data security in mobile cloud computing. *Inf. Sci.* **2017**, *379*, 42–61. [CrossRef]
6. Dropbox. Available online: <https://www.theguardian.com/technology/2016/aug/31/dropbox-hack-passwords-68m-data-breach> (accessed on 17 February 2023).
7. Arrington, M. Gmail Disaster: Reports of Mass Email Deletions. December 2006. Available online: <https://techcrunch.com/2006/12/28/gmail-disaster-reports-of-mass-email-deletions/> (accessed on 17 February 2023).
8. Amazon. Amazon s3 Availability Event: 20 July 2008. Available online: <https://simonwillison.net/2008/Jul/27/aws/> (accessed on 17 February 2023).
9. Kringsman, M. Apple's MobileMe Experiences Post-Launch Pain. July 2008. Available online: <https://www.zdnet.com/article/apples-mobileme-experiences-post-launch-pain/> (accessed on 17 February 2023).

10. Benet, J. Ipfs-Content Addressed, Versioned, p2p File System. 2014. Available online: <https://arxiv.org/abs/1407.3561> (accessed on 17 February 2023).
11. Geth. Available online: <https://geth.ethereum.org/> (accessed on 17 February 2023).
12. Metamask. Available online: <https://metamask.io/> (accessed on 17 February 2023).
13. Hao, J.; Sun, Y.; Luo, H. A Safe and Efficient Storage Scheme Based on Blockchain and IPFS for Agricultural Products Tracking. *J. Comput.* **2018**, *29*, 158–167.
14. Rajalakshmi, A.; Lakshmy, K.V.; Sindhu, M.; Amritha, P. A blockchain and IPFS based framework for secure Research record keeping. *Int. J. Pure Appl. Math.* **2018**, *119*, 1437–1442.
15. Vimal, S.; Srivatsa, S.K. A new cluster P2P file sharing system based on IPFS and blockchain technology. *J. Ambient. Intell Hum. Comput.* **2019**, 1–8. [CrossRef]
16. Chen, Y.; Li, H.; Li, K.; Zhang, J. An improved P2P file system scheme based on IPFS and Blockchain. In Proceedings of the 2017 IEEE International Conference on Big Data (Big Data), Boston, MA, USA, 11–14 December 2017; pp. 2652–2657. [CrossRef]
17. Wang, R.; Tsai, W.-T.; He, J.; Liu, C.; Li, Q.; Deng, E. A Video Surveillance System Based on Permissioned Blockchains and Edge Computing. In Proceedings of the 2019 IEEE International Conference on Big Data and Smart Computing (BigComp), Kyoto, Japan, 27 February–2 March 2019; pp. 1–6. [CrossRef]
18. Sun, J.; Yao, X.; Wang, S.; Wu, Y. Blockchain-Based Secure Storage and Access Scheme For Electronic Medical Records in IPFS. *IEEE Access* **2020**, *8*, 59389–59401. [CrossRef]
19. Ethereum. Available online: <https://ethereum.org/> (accessed on 17 February 2023).
20. Web3. Available online: <https://web3js.readthedocs.io/en/v1.8.0/> (accessed on 17 February 2023).
21. Ethers. Available online: <https://docs.ethers.io/v5/> (accessed on 17 February 2023).
22. Cai, W.; Wang, Z.; Ernst, J.B.; Hong, Z.; Feng, C.; Leung, V.C.M. Decentralized Applications: The Blockchain-Empowered Software System. *IEEE Access* **2018**, *6*, 53019–53033. [CrossRef]
23. React. Available online: <https://reactjs.org/> (accessed on 17 February 2023).
24. Krichen, M.; Lahami, M.; Al-Haija, Q.A. Formal Methods for the Verification of Smart Contracts: A Review. In Proceedings of the 15th International Conference on Security of Information and Networks (SIN), Sousse, Tunisia, 11–13 November 2022; pp. 1–8.
25. Infura. Available online: <https://infura.io/> (accessed on 17 February 2023).
26. Google. Available online: <https://www.google.com/> (accessed on 17 February 2023).
27. Facebook. Available online: <https://www.facebook.com/> (accessed on 17 February 2023).

**Disclaimer/Publisher’s Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.