A Document Contract Solution For The Modern World

Jared D. Adams
Dept. Computer Science
Colorado State University
jadams29@colostate.edu

Tom E. Cavey
Dept. Computer Science
Colorado State University
tomcavey@colostate.edu

Nicholas G. Kaufhold
Dept. Computer Science
Colorado State University
nick.kaufhold@colostate.edu

Max R. Rosoff
Dept. Computer Science
Colorado State University
mrosoff@colostate.edu

***Abstract*—In this work we evaluate the practicality of a distributed application utilizing blockchain to provide the ability to verify documents, collect signatures, and permanently preserve the record. We explore factors which influence the functionality and design of the blockchain application, such as how to sign, store, and prevent tampering of a document. The functionality is crucial in eliminating the drawbacks of using traditional document signing, as the blockchain provides security and immutability that conventional methods lack. Design is important so users understand what is happening in the background without needing to know technical details. Our goal is to design an application which makes signing documents easy, while harnessing the security of blockchain to permanently preserve the agreement. The focus of this paper is to detail the development of our application while discovering ideas that enhance the user experience and familiarity with the blockchain.**

I. Introduction and Motivation

Blockchain technology has become a frequently used buzz word in technology with the rise of digital currency. Even if you haven't been tuned in to investments and money markets you are likely to be familiar with the term Bitcoin, the famous digital currency based on blockchain technology. While this technology is the most used example of how blockchain works, digital currency isn't the only practical use of the technology.

Blockchain can simply be described as a decentralized distributed public ledger. An essential benefit of blockchain includes a ledger or record that is unable to be altered once a particular entry has been added. We've identified a use for an immutable ledger with storing contract documents that are agreements between two parties. Our aim is to ensure that both parties can be confident that the document they've signed is never altered once a signature is collected, the time stamp of agreement is never changed, and the signatories are never disputed.

A key challenge for implementing blockchain into a web application is to ensure the back end will support the features of the front end. A large element of this challenge is to create an airtight relationship between the React based UI and the Solidity contract while developing with security in mind. The nature of the application is to deal with secure documents and as such we must assume a level of fraudulent activity may be attempted. Another element of this challenge is to create an intuitive interface for the users. It is important to create the app such that any user can use the service, and instill confidence that the document and signature are being securely handled. We view both these challenges as areas that if done correctly, will set this application apart from others.

A. Goals

The focus of our project relies heavily on using blockchain technology to store a hash as well as an uploaded document. In this paper we will describe related work that we were able to leverage both Solidity code and many ideas in order to create a functioning contract signing platform for multiple users. The ultimate goal of the software is to provide the user the ability to:

- Upload a document
- Create a hash of the document (done implicitly)
- Create a contract
- List other signers of the contract
- Sign the contract

Below you will see that we were able to create an application in which all of these goals were met using multiple web development tools as well as Solidity code.

II. Related Work

According to [1] there are multiple methods for storing a document on the blockchain. One method is to create a hash and store that on the blockchain itself. Secondly, the entire document can be stored on the blockchain. Either of these methods are possible for storing the document or contract, but one of them is more popular in the industry.

Storing an entire document on the blockchain can require massive data usage, compared to what is typically stored on the chain. For example a large PDF file would need to be compressed and converted to a hexadecimal string of characters then uploaded. This may seem like a simple process, but when using a distributed system with a large number of nodes this would be an expensive and slow process. However, the benefit to this is that the document itself is stored on the chain rather than hosted elsewhere. This is key because an alternative method would require the use of a centralized database, which is counteractive to the service being distributed. Alternatively, using a distributed file system is possible, which may incur costs or add complexity. Unless the document is relatively small or is absolutely necessary to store on the chain the

following method will prove to be the most efficient in terms of blockchain resource usage. By hashing the document first and storing that in the blockchain, we are able to securely verify the document within our application of this technology.

Previous work has been conducted to create a secure document signing service using blockchain. One paper that covers some topics that apply to our project is [2], an idea was tested to protect handwritten documents which are then hashed and stored in a blockchain. This was done by taking a document of metadata and watermarking it onto the handwritten document. The meta data provides information about the handwritten document such as name, time, index, etc. Then the combined document is hashed, which is then stored on the blockchain. This watermarking procedure was shown to protect both records, as an alteration to one or either file will result in the hashes not matching. We found this applicable to our work in solving a document storage problem, which we have approached in a similar strategy. Creating a hash of the document and storing it on the chain rather alleviates the need for an additional location to store a copy of the signed document. If the document was a legal contract and the metadata was a signature it would be possible to verify that a specific person signed the document, if their signature can be validated in some way.

According to these articles [3] and [4] legislation has been passed in multiple states that allow signatures on the blockchain to be admissible in court, therefore making it a legally binding contract. The application of services similar to Sign Safe may become increasingly more important as the shift from contracts to digital smart contracts occur. We believe this technology will provide more transparency and better immutability than traditional contracts.

III. Technical Results

Creating and using a hash of the document on the blockchain is much more efficient when compared to storing the entire document. Using the Secure Hash Algorithm, or SHA for short, we can take the document and run it through the cryptographic hash like SHA-256 to create the hexadecimal hash string. This string is then stored on the blockchain rather than the document itself. The document could be stored in a centralized server or a distributed file system such as IPFS (which is what we used) and accessed later at any time, but is not necessary if the document is publicly available. The reason this works best is because storing the hash of the document on the chain requires less resources and takes up less space than a large document would. The document hash will not match if someone has attempted to alter the document, so the authenticity must always be evaluated by looking at the hashes. The downside of the hashing method is not being able to store the document where it is readily available. For our project we have omitted this piece for later work and assume all parties have the document on their local system. The main purpose of using blockchain in our application is to store only the hash. This is much more efficient from a resource standpoint, however the drawback may be enough that others with extremely important or sensitive documents will prefer the alternative.

For our application a user will go to the link, http://www.signsafe.technology/. Any user of the application will be immediately prompted to sign in with their Metamask account. After signing in they will see the options to either create a new contract or enter a contract web link that has already been created. The former is used to create a new contract to be signed, the latter for additional signers of a contract. On contract creation, a user is prompted to submit their name, email, and Metamask account address in which they want to use to create a contract. Then they are prompted to submit additional names and emails of the other signers. Next they will upload a document that is hashed. Finally, the contract will be submitted onto the Rinkeby test network via verification and payment of gas through Metamask. As for signers of a contract, they will enter the link to the contract (auto generated by the application) and will be requested to upload the same document that was hashed earlier. If the hashes do not match, then the contract cannot be signed and will prompt the user they do not have the correct document. If the hashes do match, Metamask will prompt them to pay for the gas cost and the contract will be signed.

We were able to use the development platform Netlify in order to host our application. From this for every contract created we generated at random a sequence of characters as an additional piece to the link such as '/jcqgkw9x' in which is the actual web address of that specific contract. So in full: 'http://www.signsafe.technology/jcqgkw9x'. The document can be seen for the duration of contract signing after upload and is stored temporarily on the IPFS such that for web page transition the user can visually see their document. The hashing of the document is done by SHA256 which is the standard hashing method for Solidity. This hash is computed for both the initial upload and any additional signer who uploads the document. The two hashes are compared in order to verify they are the same and once they are verified that's when Metamask will request funds in order to sign the document.

Our solidity code is divided into two solidity files: SignSafeContract.sol and MultiPartyContract.sol. The first is SignSafeContract.sol and is set up as a parent is inherited from for other types of contracts. The MultiPartyContract.sol is our implementation of uploading and signing a contract on the blockchain.

'SignSafeContract.sol' contains several variables such as the contract_owner, contract creation_date, contract_completed_date, and contract_hash. We anticipated that these variables would be essential for any type of contract.

The SignSafeContract.sol allows our contracts to be in four possible states: SETUP, PENDING, COMPLETED, and CANCELLED. The SETUP state is for the initial creation of the contract such as uploading the contract hash and adding signatories. At this stage, no one is permitted to sign the contract. Once the contract is set up, it is moved into the

PENDING state, which is used when the contract can be signed by the parties. Finally, if all of the parties have signed the contract, it will enter the COMPLETED state, or, if one of the parties has rejected the contract it will enter the CANCELLED state.

In order to store data about the actions of the signatories, we utilize four mappings. First, we have a mapping for who has signed, which keeps track of which addresses have and have not signed the contract. Second, we have a mapping for the signatories so that we can limit signing to only those specified by the contract creator. Third, we have a mapping labeled 'hasMatch' that stores a boolean for each address that has a matching hash of the contract. Finally, we have a mapping for the time that each signatory signed the contract.

Lastly, SignSafeContract.sol contains modifiers, events, and two functions that we believed that all contracts should have access to. The modifiers are used to limit actions depending on the state of the contract. For example, adding signatories or changing the contract should only occur during SETUP, otherwise a nefarious actor could change the content of the contract after the parties had signed. The events are used to record key events on the blockchain such as the date a contract is submitted, signed, cancelled, or completed. The functions in SignSafeContract.sol give the contract owner the ability to cancel the contract if it is not completed and to confirm that a signatory has a hash of the contract. Requiring a signatory to have a matching hash of the contract gives an extra layer of security and confidence in the contract because it means that the signer has an exact copy of the contract they are signing. In our implementation we chose to do this in the user interface by calling getContractHash to obtain the hash that is stored in the blockchain.

The MultiPartyContract.sol is the actual contract that we used in the implementation of our website. MultiPartyContract.sol inherits from SignSafeContract.sol so it has all of the functionality described above. This contract is designed to have as many parties to a contract as the contract creator desires (so it is not limited to only two parties).

The bulk of the work for MultiPartyContract is done in the constructor. The constructor takes a string of the contract hash (which is created in the user interface) and an array of addresses that contains the address for each signatory. The constructor then does all of the work of the SETUP phase and puts the contract directly into the PENDING state. There are several functions in the class that became irrelevant as we adjusted the constructor to perform all this work (addSignatory, setNumberOfParties, deleteSignatory, and contractReadyForSignatures), but we opted to leave those methods so they could be used in future contracts that might inherit from MultiPartyContract.sol.

```
constructor (string memory _contractText,
  address[] memory _signatoryAddresses) public{
    contract_owner = msg.sender;
    STATE = sign_safe_contract_state.PENDING;
    numberOfSignatures = 0;
    numberOfSignatoriesAdded = _signatoryAddresses.length;
    numberOfParties = _signatoryAddresses.length;
    creation_date = now;
    contractHash = _contractText;
    contractUploaded = true;
    for(uint i = 0; i < _signatoryAddresses.length; i++){
        signatories[_signatoryAddresses[i]] = true;
    }
    emit contractCreated(contractHash, creation_date);
}
```

After the MultiPartyContract is created, a signatory (and only a signatory that is contained in the mapping) can perform one of two actions. First, if the signatory has the option to cancel the contract. If a user does this, a message is emitted to the blockchain and the state is changed to CANCELLED and no one else can sign the contract. Alternatively, a user can sign the contract, which will emit a message to the blockchain that records who signed the contract and the date of the signature. If the signer is the last person to sign, then the contract state is changed to COMPLETED and an event is emitted to the blockchain that records the date the contract was completed.

```
function sign() only_PENDING only_unsigned_signatories
  only_signatories public {
    numberOfSignatures++;
    who_has_signed[msg.sender] = true;
    signature_timestamps[msg.sender] = now;
    emit signature(msg.sender, true, now);
    if(numberOfSignatures == numberOfParties){
        completeContract();
    }
}

function completeContract() only_PENDING private {
    STATE = sign_safe_contract_state.COMPLETED;
    contract_completed_date = now;
    string memory emit_message =
      "All signatories have signed the contract. ";
    emit contractComplete(emit_message, true, now);
}
```

IV. Future Work

There are various improvements to be made that we have purposefully left out due to time constraints. The application will most certainly require notifying all signers via email about where to go in order to sign the contract(s). This is a crucial aspect that will make this application fully capable of being a common resource for contract signatories. Another aspect left to be worked on is the capability of a document to be stored on the web such that users will not need to have the same document on their local computer. Something such as storing the document on a database that can pull the document onto the application once other signers click on the web link

sent to them will ultimately let this application become very standalone. Finally, bug testing is always necessary in any software development and is a continuous process. Otherwise, the application currently has met the criteria of our set goals and is purposeful in its current design.

V. Conclusion

Overall, as a team we have created a fully functioning application that enables a user with a Metamask account to create a contract on the Rinkeby test network with the intent on signing a contract with other users that's placed on a blockchain. Furthermore, this project has displayed how very simplistic the code is that is necessary to perform this task. There will always be the option of adding to the application itself but as a team the ability to create a working application such as this shows the versatility of blockchain technology and how easy it is to use and understand.

References

[1] D. Francati, G. Ateniese, A. Faye, A. M. Milazzo, A. M. Perillo, L. Schiatti, and G. Giordano, "Audita: A blockchain-based auditing framework for off-chain storage," 2019.

[2] R. Latypov and E. Stolov, "A new watermarking method to protect blockchain records comprising handwritten files," 2019.

[3] S. Higgins, "Arizona governor signs blockchain bill into law," 2017. [Online]. Available: https://www.coindesk.com/arizona-governor-signs-blockchain-bill-law

[4] ——, "Vermont is close to passing a law that would make blockchain records admissible in court," 2016. [Online]. Available: https://www.coindesk.com/vermont-blockchain-timestamps-approval